

Phasen eines SE-Projekts

- vor Projektauftrag kommt Projektantrag
- Inhalt Projektantrag: Ausgangslage, Ziele, Kosten, Nutzen, Organisation
- Inhalt Projektauftrag: Bezeichnung, Auftraggeber, Beginn / Ende, Beschreibung, Budget, Team
- Externe Vergabe: Ausschreibung → Angebote → Auftragsvergabe
- Phasen:
 - o Analyse: Systemspezifikationen
 - o Design: Technisches Design
 - o Implementierung und Modultests: Programmcode, Testberichte ...
 - o Test und Integration: auf produktnahe Umgebung, Testberichte
 - o Abnahme und Einführung: Abnahmeprotokolle, Testprotokolle
 - o Betrieb und Wartung: längste Phase, Fehler beheben & System verbessern

Aufwandsschätzung

- Ansätze:
 - o Expertenschätzung:
 - Fachleute mit Erfahrungen schätzen Aufwand
 - Einzelschätzung
 - Mehrfachbefragung: Durchschnitt aus unabhängigen Abfragen
 - Delphi-Methode:
 - unabhängige Schätzung, Aggregiertes Ergebnis zurück an Experten
 - Verfeinerung durch Experten bis Ergebnis stabil
 - Schätz-Klausur: Kollektivschätzung mehrerer Experten
 - o Algorithmische Schätzung:
 - Kosten aus Größen berechnen, die frühzeitig bekannt sind
 - COCOMO:
 - Formeln aus archivierten Daten, Aufwand und Entwicklungsdauer damit schätzen
 - Berücksichtigung besonderer Umstände durch Einflussfaktoren
 - Function Point: Programm mit verschiedenen Einflussfaktoren
 - Basis für die Schätzung ist der Umfang des Programms in zu implementierenden Funktionen

Requirements Engineering

Hauptaufgaben:

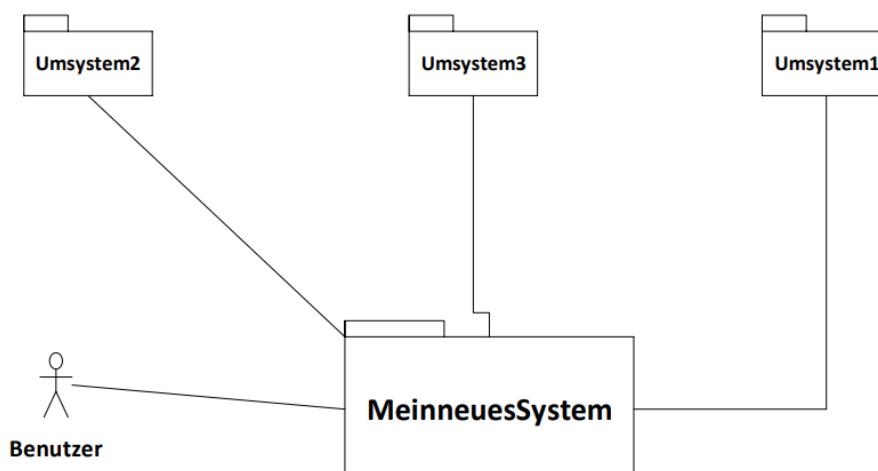
- Analysephase zur frühzeitigen Fehlererkennung und -behebung spart Zeit, Kosten und Ärger
- RE hilft beim zusammenstellen der Anforderungsspezifikationen (Lastenheft & Pflichtenheft)
 - o Anforderungen ermitteln, spezifizieren, analysieren und validieren
- Aufbau eines Requirement Dokumentes
 - o **Einleitung**
 - Zielsetzung, Produktziele, Referenzen, Überblick
 - **Übersichtsbeschreibung**
 - Produktumgebung & -funktion, Restriktionen, Annahmen und Abhängigkeiten
 - o **Spezifische Anforderungen**
 - externe Schnittstellen-, funktionale, Leistungsanforderungen, Eigenschaften des Softwaresystems

Objekte des RE:

- Visionen und Ziele formulieren (oft in Projektauftrag und -antrag)
 - o Vision ist realitätsnahe Vorstellung der Zukunft, Ziele zum Verfeinern der Visionen
 - o kurz, prägnant und überprüfbar formulieren, Mehrwert hervorheben, begründen, realistisch formulieren, keine Lösungsansätze
- Rahmenbedingung: organisatorische und technische Restriktion für das System & die Entwicklung
 - o organisatorisch: Anwendungsbereiche, Zielgruppen
 - o technisch: Produkt-/Entwicklungsumgebung
- **Anforderungen:** **welche Eigenschaften werden von SS erwartet**
 - o funktional: Funktionen, Dienste, Reaktion auf Eingaben, Verhalten in best. Situationen
 - o Leistung: Zeit, Geschwindigkeit, Umfang, Durchsatz
 - o Qualität: **Zuverlässigkeit, Benutzbarkeit, Sicherheit, ...**
 - o Randbedingungen: Physikalisch, Rechtlich, Kulturell, Schnittstellen, ...
- **Qualitätsmerkmale:** Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Wartbarkeit, Portabilität
- Anforderungen sollen korrekt, eindeutig, verständlich, klassifizierbar, überprüfbar formuliert werden

Tätigkeiten des RE:

- Systemkontext festlegen:
 - o Teil der Umgebung des Systems, der für die Definition und das Verständnis relevant ist
 - o System von der Umgebung abgrenzen, Anforderungen relevanter Umgebungsteile identifizieren
 - o Aspekte: Stakeholder, System im Betrieb, Prozesse, Dokumente
 - Stakeholder: Person oder Organisation mit Einfluss auf die Anforderungen
 - Quellen für Anforderungen
 - z.B. Kunde, Zulieferer, Gesellschaft, Eigentümer, Mitarbeiter
 - o **Kontextdiagramm:** Beschreibung des Umfelds vom System



- Anforderungen erfassen:
 - o Anforderungen möglichst vollständig und fehlerfrei aufnehmen
 - o Quellen: Stakeholder, Dokumentation, System im Betrieb
 - o Einflussfaktoren: bewusste & unbewusste Anforderungen ermitteln, Termine, Budget, Chancen & Risiken
 - o Ermittlungstechniken: Befragungstechniken, Dokumentgetrieben, Kreativitätstechniken, Beobachtung
 - o Für jede Anforderung: ID, Name, Beschreibung, Version, Autor, Begründung, Quelle, Abnahmekriterien

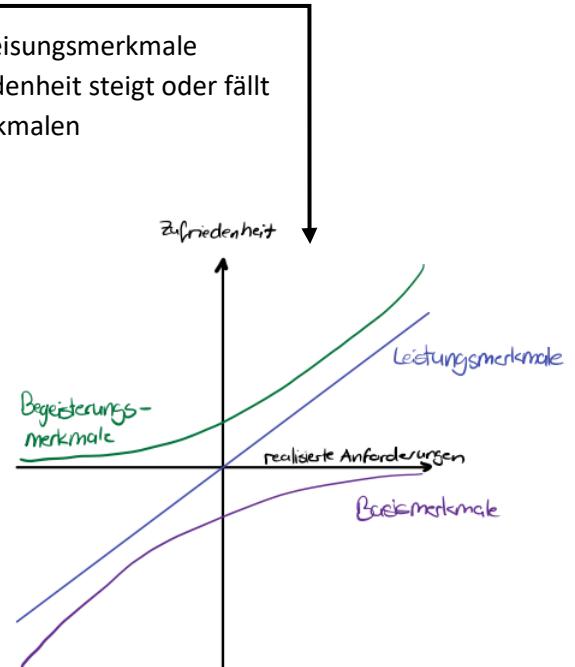
- Anforderungen dokumentieren:
 - o rechtliche Relevanz, Ausgangspunkt für Design, Test und Abnahme, zur Verfügung für alle Beteiligten
 - o Lastenheft: grobes Pflichtenheft, vom Auftraggeber, lässt Details offen
 - Visionen & Ziele, Rahmenbedingungen, Kontext & Überblick, Funktionale und Qualitätsanforderungen
 - o Pflichtenheft: vom Auftragnehmer, detailliert lieferndes System, entspricht Gesamtspezifikation
 - alles aus Lastenheft detaillierter und Abnahmekriterien
 - o besteht in der Regel aus Prosa und Modellen
 - o Modellierungssprachen: Use cases, Aktivitätsdiagramme, Klassendiagramme, Zustandsdiagramme
 - o Perspektiven der Anforderungsdokumentation:
 - Strukturperspektive UML-Klassendiagramme
 - Funktionsperspektive UML-Aktivitätsdiagramme
 - Verhaltensperspektive Statecharts

- Anforderungen prüfen und abstimmen
 - o Analyse: Prüfung gegen Qualitätsanforderungen
 - Inspektionen, Walkthroughs, Stellungnahme, Round Robin Review, Peer Review
 - Kano Modell:
 - Basis-, Leistungs-, Begeisterungs- Unerhebliche und Rückweisungsmerkmale
 - Je nach Erfüllung der einzelnen Merkmale → Kundenzufriedenheit steigt oder fällt
 - Begeisterungs- werden zu Leistungs- und dann zu Basismerkmalen
 - o Validierung: Prüfung gegen Visionen & Ziele

- Anforderungen verwalten
 - o toolgestützt
 - o Änderungen, Nachverfolgbarkeit, Versionierung, Zustandsmodell

Risiken:

- unzureichende Repräsentation von Kunden
- übersehen von kritischen Anforderungen
- unkontrollierte Änderung von Anforderungen
- keine Qualitätsprüfung
- Perfektionierung von Anforderungen



Best Practices:

- Kunden & Benutzer einbinden
- Erfahrene Projektmanager und Teammitglieder einsetzen
- Anforderungen priorisieren
- Anforderungsschablonen und Beispiele zur Verfügung stellen
- ...

Objektorientierte Analyse

Definition:

- Ermittlung und Beschreibung der Anforderungen an ein SS durch objektorientierte Konzepte und Notationen
- Basiskonzepte:
 - o Objekt: Gegensand des Interesses
 - o Klasse: definiert Struktur (Attribute) und Verhalten (Operationen) für gleichartige Objekte
 - o Attribut: Daten, die von Objekten einer Klasse angenommen werden können
 - o Operation: Ausführbare Tätigkeit des Objekts
- UML: Unified Modelin Language
- o grafische Darstellungsform zur Visualisierung, Spezifikation, Konstruktion und Dokumentation von SS
- o Analysephase:
 - Use case Diagramm: Analysephase
 - welche Use Cases sind enthalten, wer löst sie aus, welche Abhängigkeiten bestehen?
- o Analyse und Designphase:
 - Aktivitätsdiagramm:
 - Welche Schritte werden innerhalb eines Use Cases durchlaufen?
 - Paketdiagramm:
 - Pakete in der Anwendung, welche Kommunikation zwischen Paketen muss realisiert werden?
 - Klassendiagramm:
 - Zusammenhänge in der Aufgabenstellung, Klassen, Komponenten, Pakete, Methoden & Eigenschaften
 - Sequenzdiagramm:
 - Methoden für Kommunikation zwischen Objekten, zeitlicher Ablauf von Mehtodenufrufen
 - Kommunikationsdiagramm:
 - Kommunikation zwischen Objekten, Reihenfolge von Methodenufrufen
 - Zustandsdiagramm:
 - durch Methoden ausgelöste Zustandsübergänge, Zustand nach Erzeugung eines Objekts
- o Designphase
 - Komponentendiagramm:
 - Kapselung von Soft- und Hardware mit Funktion, Interfaces zwischen Komponenten
 - Einsatz- und Verteilungsdiagramm:
 - welche Module auf welcher HW, auf welcher Kommunikationsmöglichkeit basiert Zusammenarbeit

Makroprozess:

1. relevante Geschäftsprozesse ermitteln
 2. Klassen ableiten
 3. statisches Modell erstellen
 4. dynamisches Modell parallel zu 3. erstellen
 5. Wechselwirkung beider Modelle berücksichtigen
- Alternativen:
 - o Szenariobasierter Makroprozess:
 - Geschäftsprozesse formulieren
 - Szenarios aus den Geschäftsprozessen ableiten
 - Interaktionsdiagramme ableiten
 - Klassen- und Zustandsdiagramme erstellen
 - o Datenbasierter Makroprozess:
 - Klassendiagramme erstellen
 - Geschäftsprozesse formulieren
 - Szenarioas aus den Geschäftsprozessen ableiten
 - Interaktionsdiagramme aus den Szenarios und dem Klassendiagramm ableiten
 - Zustandsdiagramme erstellen

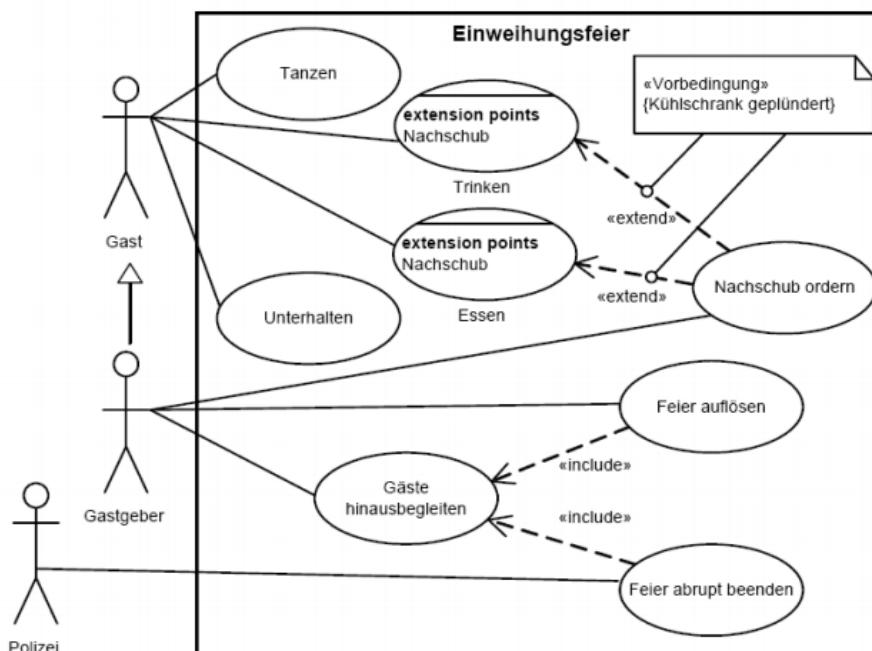
- Aufgabenbereiche:
 - o Analyse im Großen:
 - Use Case Modell aufstellen: Diagramm, Beschreibungen, Aktivitäts- und Zustandsdiagramme)
 - Pakete bilden: (Teilsysteme bilden, Paketdiagramm)
 - o statisches Modell:
 1. Klassen identifizieren
 2. Assoziationen identifizieren
 3. Attribute identifizieren
 4. Vererbungsstrukturen identifizieren
 5. Assoziationen Vervollständigen
 6. Attribute spezifizieren

(Klassen- & Objektdiagramme)
 - o dynamisches Modell:
 1. Szenarios erstellen
 2. Zustandsdiagramme erstellen
 3. Operationen eintragen
 4. Operationen beschreiben

(Sequenz-, Kollaborations-, Zustands-, Klassen & Aktivitätsdiagramme)

Use-Case-Diagramm:

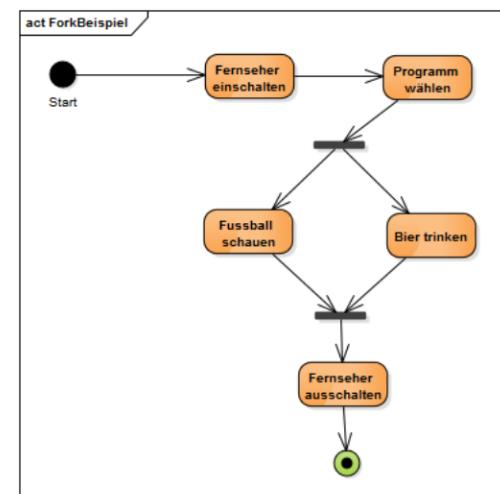
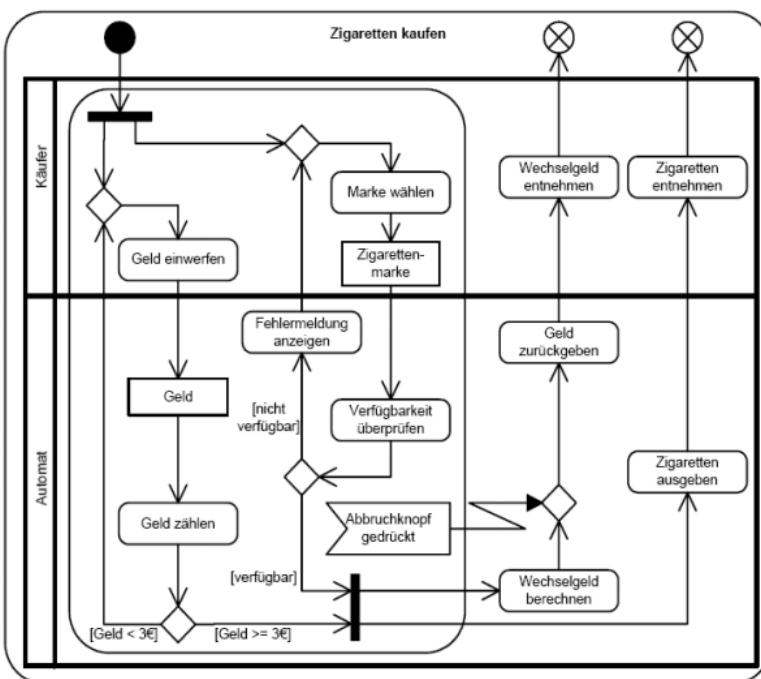
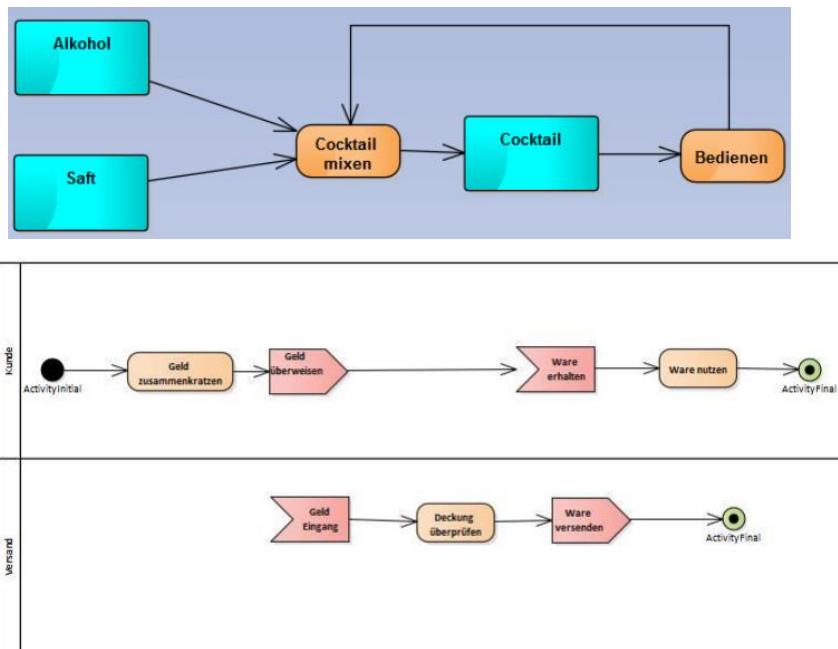
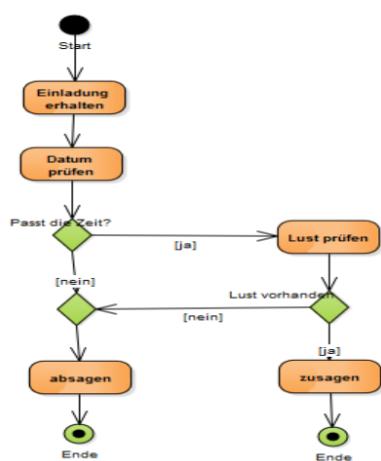
- Definition:
 - o Beschreibung der einzelnen Schritte zur Zielerreichung
 - o konkrete technische Lösungen abstrahieren
 - o aus externer fachlicher Sichtweise beschrieben
 - o zeigt Absicht des Akteurs, nicht das Verhalten des Systems
- Struktur-Ebene / Use-Case-Diagramm:
 - o zeigt externes Verhalten eines Systems gegenüber seinen Akteuren
 - o zeigt strukturelle Abhängigkeiten zwischen den Anwendungsfällen und Akteuren
- Detaill-Ebene / Use-Case-Definition:
 - o natürlichsprachliche Beschreibung der Arbeitsabläufe, also der Interaktion von Akteuren mit System
 - o ergänzt um Zustands- und Aktivitätsdiagramme
 - o Inhalte:
 - Name, Kurzbeschreibung, Akteure, Auslöser, Eingehende Daten, Vorbedingungen, Nachbedingungen, Ergebnis, Essentielle Schritte, Alternativszenarien, Offene Punkte, Änderungshistorie
- erlauben eine „Black Box“-Sicht auf das System, unabhängig von der technischen Realisierung
 - o Abgrenzung des Systems von der Umwelt
- Einsatzbereich:
 - o funktionale Dienstleistungen eines Systems auf einen Blick zeigen
 - o aus Nutzersicht in handhabbare, logische Teile zerlegen
 - o Außenschnittstellen und Kommunikationspartner des Systems modellieren
 - o komplexe Systeme leicht verständlich darstellen

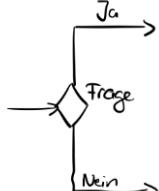
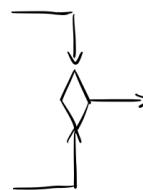
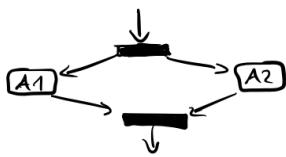
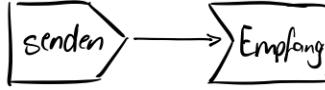
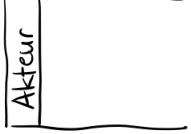
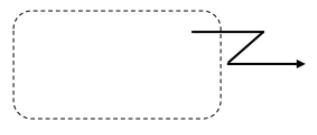


Notation	Name	Bedeutung
	Use-Case	fachlicher Anwendungsfall, Verhalten beschreiben ohne auf Details einzugehen
	Akteur	externes Objekt oder Person, die mit dem System interagiert (startet einen Use-Case)
	Akteur	Fremdsystem als Akteur
	Akteur	Zeitgesteuertes Ereignis
	Ableitung	Spezialisierung / Generalisierung von Akteuern oder Use-Cases
	Assoziation (ungerichtet)	Akteur ist an einem Use-Case beteiligt bzw. System kommuniziert mit Use-Case
	Assoziation (gerichtet)	Beschreibt welcher Teil der aktive ist
	Multiplizität	Beschreibt, wie oft das entsprechende Element gleichzeitig an der Transaktion teilnehmen kann
	include	ein Use-Case (A) importiert Verhalten von Use-Case (B)
	extend	(A) kann von (B) erweitert werden, der Extension Point gibt an, ab wann erweitert wird

Aktivitätsdiagramm:

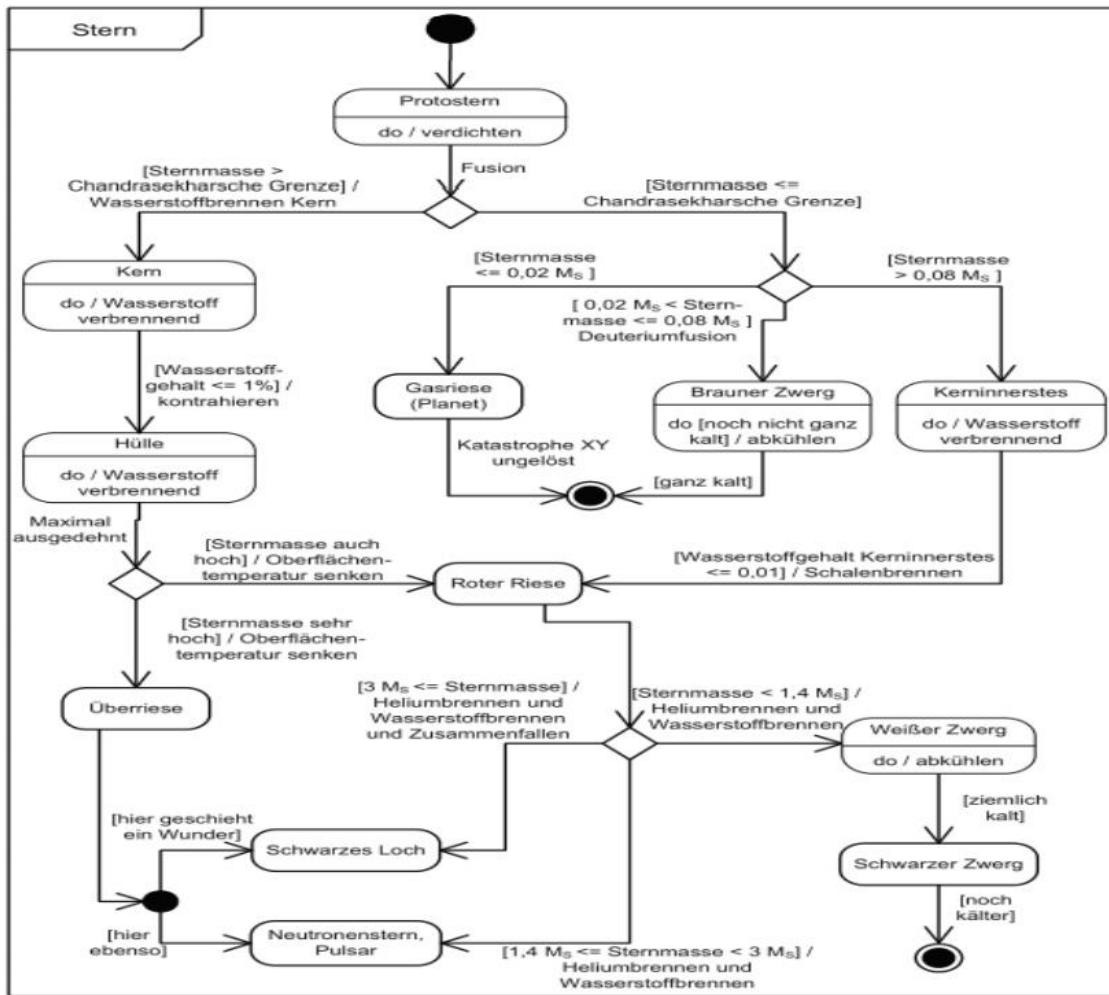
- Abläufe und Prozesse in Zusammenhang mit einem System modellieren
- können sequentielle und parallele Abläufe modellieren
- Beispiele zur Verwendung:
 - o Abarbeitung eines Use-Cases
 - o Algorithmus einer Operation
- Kanten:
 - o Kontrollfluss: zwischen 2 Aktionen, Token ohne Daten nur zur Aktionsausführung
 - o Objektfluss: Kante mit min. 1 Objektknoten, Token trägt Daten
 - o Kanten können mit Bedingungen belegt werden
 - o Kanten können durch Sprungmarken unterbrochen und an anderer Stelle fortgeführt werden
 - o PinNotation: Ausgabe einer Aktion ist Eingabe der nächsten (Pin an Anfang und Ende der Kante)



Notation	Name	Bedeutung
●	Startknoten	Startpunkt, mehrere Startknoten erlaubt, alle Startknoten beginnen gleichzeitig
○	Endknoten	beendet beschriebenen Ablauf, min. 1 Endknoten / Diagramm
Action	Action	zentrales Element, modelliert einen einzelnen Schritt, über Kanten mit anderen Elementen verbunden
Bedingung	Vor- und Nachbedingungen	bei Aktionsstart und -ende zu beachten
⊗	Ablaufende	beendet einzelnen Kontrollfluss
	Entscheidung / Decision	Verzweigung eines Kontrollflusses, ein oder mehrere ausgehende Kanten mit Bedingungen, ein Eingang
	Zusammenführung / Merge	mehrere eingehende Kanten zu einer Kante
	Teilung / Fork Synchronisation / Join	Kontrollfluss wird geteilt Kontrollfluss wird zusammengeführt
	Signal	Aktion, die ein Signal sendet
	Swimlane	Beschreibt, wer oder was für eine Menge von Knoten verantwortlich ist (z.B. Abteilung)
Auftrag (Zustand)	Objektknoten	Ergebnis von Action kommt rein, Eingabe für nächste Action geht raus
	Aktivität	verkürzte Schreibweise: Eingabeparameter gehen rein, Ausgabeparamter gehen raus
	Unterbrechung	Unterbrechung wie Exceptions modellieren, verlässt man Bereich über Unterbrechungskante werden sämtliche Aktionen unterbrochen & Tokens verworfen

Zustandsdiagramm:

- Verhalten von Klassen, abhängig vom internen Zustand der betrachteten Klasse
- werden verwendet wenn die Reaktionen einer Klasse direkt von internen Zuständen abhängig sind
- Trigger können Signale, „CompletionEvents“ (wenn do-Aktion von Ausgangszustand beendet ist), TimeTriggers und Operationen (when $x > y$, Call-Events ...) sein
- wird für jeden nicht trivialen Lebenszyklus erstellt



Zustandsdiagramm

Notation	Name	Beschreibung
<pre> Zustand entry : Eintrittsaktion do : Aktion exit : Austrittsaktion </pre>	Zustand	<ul style="list-style-type: none"> - bildet Situation ab, in deren Verlauf spezielle Bedingungen gelten - Regel: <ul style="list-style-type: none"> o wird aktiv beim Betreten o beim Betreten : entry - Aktion o danach do - Aktion o beim Verlassen inaktiv & exit - Aktion
 <u>Trigger [Guard]/Aktivität</u>	Transition	<ul style="list-style-type: none"> - Übergang von Ausgangs - in Zielzustand - Transition wird durchlaufen, wenn Trigger-Ereignis stattfindet - Durchlauf nur wenn Guard-Bedingung = true
	Startzustand	Startpunkt für Zustandsautomat (nur 1x)
	Endzustand	Endpunkt
 <u>[Bed. 1]</u> <u>[Bed. 2]</u>	Kreuzung	<p>erst Entscheidung, dann Triggeraktion ausführen,</p> <p>z.B. </p>
 <u>[Bed. 1]</u> <u>[Bed. 2]</u>	Entscheidung	<p>erst Trigger ausführen, dann Entscheidung</p> <p></p>
	Terminator	- Abbruch des Automaten, Lebensdauer des Classifiers beenden
 <i>mögl. direkt Zustand ansprechen</i>	zusammengesetzter Zustand	<p>zeigt innere Struktur, Default Entry, Explicit Entry</p> <p>Ausgang je nach Definition mit Trigger, nach Zustand oder default</p>
	fork <i>immer paarweise!</i>	<p>keine Guards auf Ausgängen, aber versch. Verhalten erlaubt</p> <p>keine Guards auf Eingängen erlaubt</p>
	Region	<ul style="list-style-type: none"> - Trigger werden in allen Regionen gleichzeitig beachtet - Default und explicit entry

Paket Diagramm

- Wie kann ich mein System so einstellen, dass ich den Überblick behalte?
- Ein Paket entspricht einem in sich geschlossenen Teilsystem, können geschachtelt werden

Notation	Name	Beschreibung
<div style="border: 1px solid black; padding: 5px; width: fit-content;">Paketname</div>	Paket	<ul style="list-style-type: none"> - fasst paketierbare Elemente zusammen - definiert einen Namensraum
<p>-- - << Ausdruck >> --></p>	Importpfeil	<p><<import>>: public import</p> <p><<access>>: private import</p> <p><<merge>>: anlegen von neuen Objekten beim Import</p>

Schritte zum Identifizieren von Paketen:

- Top Down: vor der Erstellung der Use Cases
- Bottom Up: nach Erstellung der Geschäftsprozesse oder nach der statischen Modellierung

Analyse:

- Analytische Schritte:
 - bildet Paket eine abgeschlossene Einheit?
 - Ist der Paketname geeignet

Klassendiagramm

Ziel: Übergang von Problem zu Implementierung ohne Bruch

↳ Domänenmodell: Problembereich modellieren, Wissen des Problembereichs als abstraktes Modell darstellen

- Informationen:
- Existenz von Klassen
 - Attribute der Klassen
 - Operationen der Klassen
 - Beziehungen zwischen Klassen

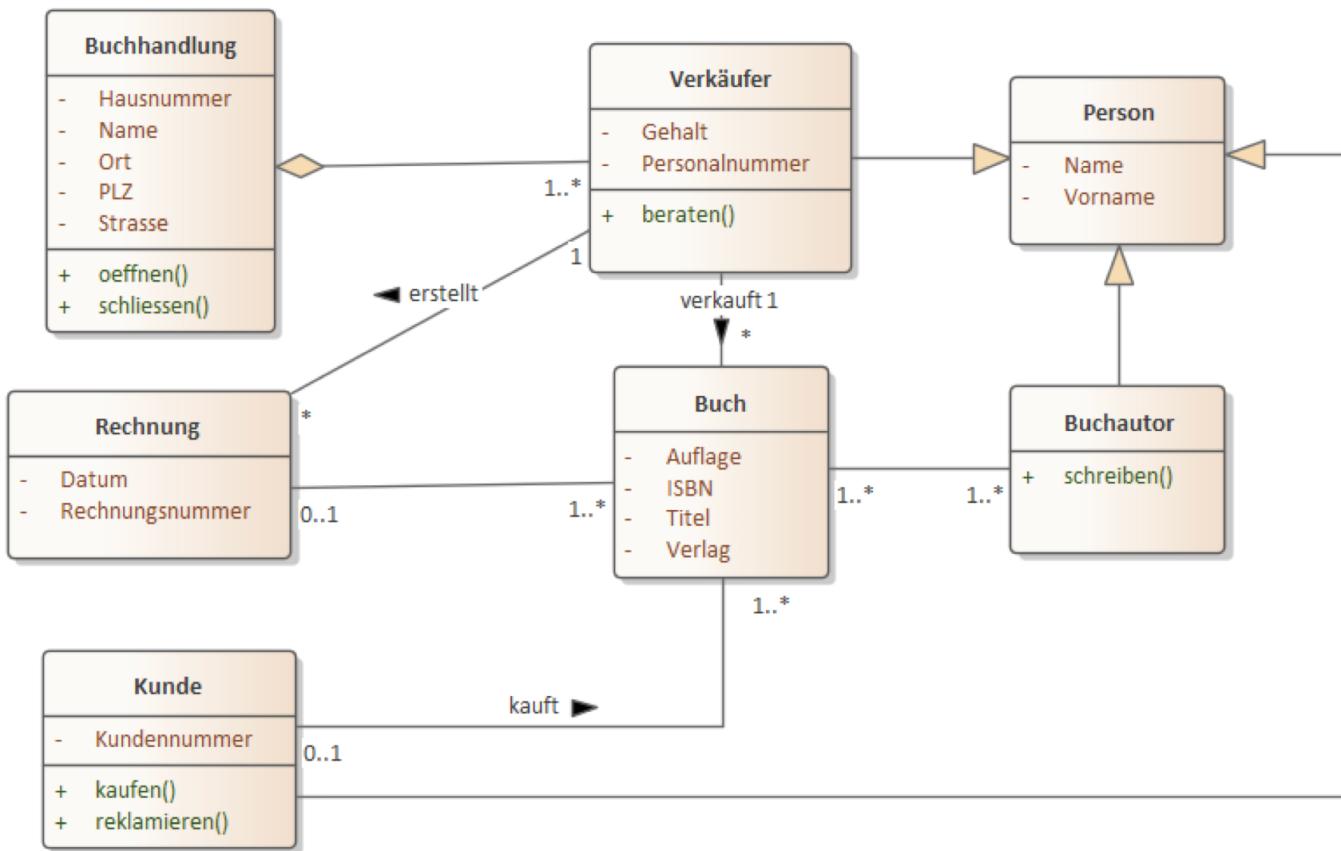
Notation	Name	Beschreibung			
<table border="1"> <tr><td>Klassename</td></tr> <tr><td>- Attribute</td></tr> <tr><td>+ Operationen()</td></tr> </table>	Klassename	- Attribute	+ Operationen()	Klasse	<ul style="list-style-type: none"> - ohne Fremdschlüssel & Verwaltungsoperationen (Getter & Setter ...) - richtiges Abstraktionsniveau - aussagekräftiger Klassename
Klassename					
- Attribute					
+ Operationen()					
<p>+ Rolle Navigationsrichtung + Rolle * name 1... Multiplizität</p>	Assoziation	<ul style="list-style-type: none"> - Zusammenhänge zwischen Klassen - Beziehungen zwischen den Objekten der Klassen 			
<p>Ganzes + Rolle Teil 0..* 1..*</p>	Aggregation	schwache Abhängigkeit, Teil kann auch ohne Ganzes existieren (z.B. Geschäft - Verkäufer)			
<p>Ganzes + Rolle Teil 1..* 1..*</p>	Komposition	starke Abhängigkeit: Teil kann ohne Ganzes nicht existieren (z.B. Buch - Seite)			
	Vererbung	<p>"ist ein", Übernahme von Parametern</p> <p>↓ Spezialisierung ↑ Generalisierung</p>			
<p>Vorlesung Person Teilnahme</p>	Assoziationsklassen	<p>\cong Vorlesung $\xrightarrow{\text{Teilnahme}}$ Person</p> <ul style="list-style-type: none"> - Eigenschaften definieren, die nicht direkt einem der Partner zugesiesen werden können 			

Definition von Operationen:

[Sichtbarkeit] Name (Parameterliste) : Rückgabetyp [{Eigenschaftswert}]*

+ public
 - private
 # protected
 ~ package

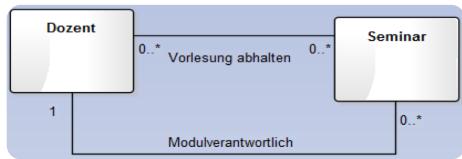
in
 out + wie Attribute
 inout



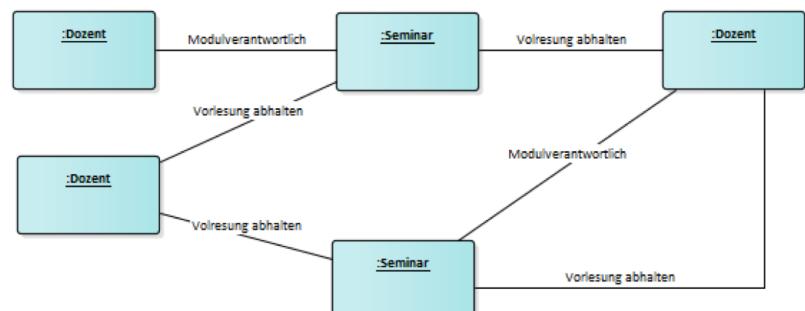
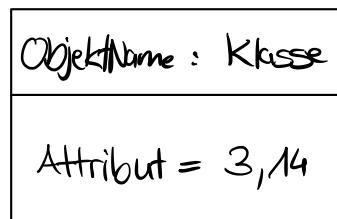
Objektdiagramme

- Momentaufnahme vom System darstellen
- Umwandlung der Elemente vom Klassendiagramm
 - ▷ Klasse → Instanz / Objekt
 - ▷ Assoziation → Link
 - ▷ Attribut → Wert / Slot
- zur Prüfung von Klassendiagrammen und Illustration rekursiver Strukturen

Klassendiagramm



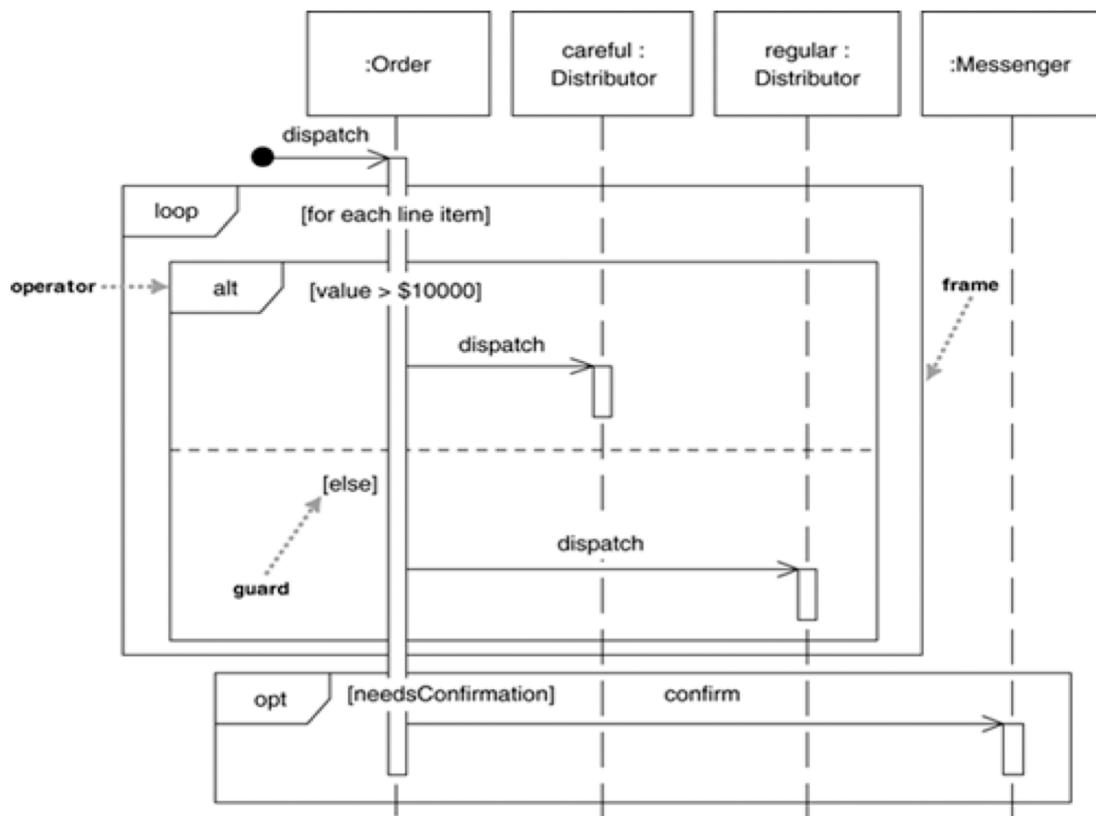
Objektdiagramm



Sequenzdiagramm

- Kommunikationsablauf im System darstellen
- am häufigsten verwendetes und mächtigstes Interaktionsdiagramm
- zeitlicher Ablauf von oben nach unten, Kommunikationspartner von links nach rechts

Notation	Name	Beschreibung
<p>Objektname: Klasse</p> <p>methode(): rückgabe</p>		<p>zentrale Struktur</p> <p>fork Diagramm</p>
	synchroner Aufruf	Empfänger verarbeitet Nachricht und liefert Antwort (<--) zurück
	asynchroner Aufruf	Quelle wartet mit der weiteren Verarbeitung nicht auf die Antwort (kein <--)
<p>testen()</p>	geschichtelter Aufruf	
<p>: Seminar</p> <p>: Login</p>	Objekt kreieren	
<p>Object zerstören</p>		
<p>operator!</p> <p>[Bed 1]</p> <p>[Bed 2/ esc]</p>	Operatoren	<p>alt: Alternativen (if / else)</p> <p>opt: Optional (wenn Bed = true)</p> <p>par: parallele Ausführung</p> <p>loop: Schleife</p> <p>region: kritische Region (nur 1 Thread erlaubt)</p> <p>neg: negativ (nicht erlaubte Interaktion)</p> <p>ref: Verweis auf ein anderes Diagramm</p>

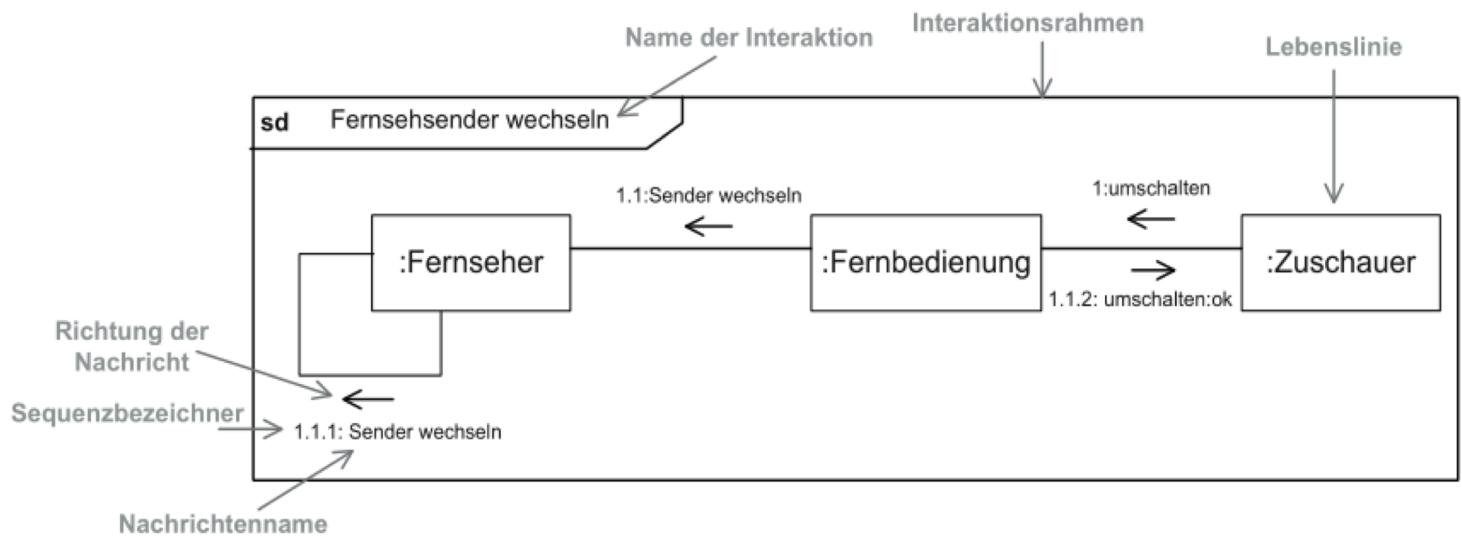


Verwendung:

- im Feindesign zur Feinmodellierung von Funktionsaufrufen
 - ↳ Interaktion / Kommunikation zwischen Objekten in einer bestimmten Szene
- Analysephase: zur Abgrenzung der Kontexte, wenn Interaktionsverhalten nicht genau definiert ist, Unterstützung bei Usecase Modellierungen
- Architektur: Komponentenkommunikation / Schnittstellenmodellierung

Kommunikationsdiagramme:

- Wechselspiel und Nachrichtenaustausch von Teilen einer komplexen Struktur
- Zeit steht nicht im Vordergrund



Sequenzbezeichner : Name

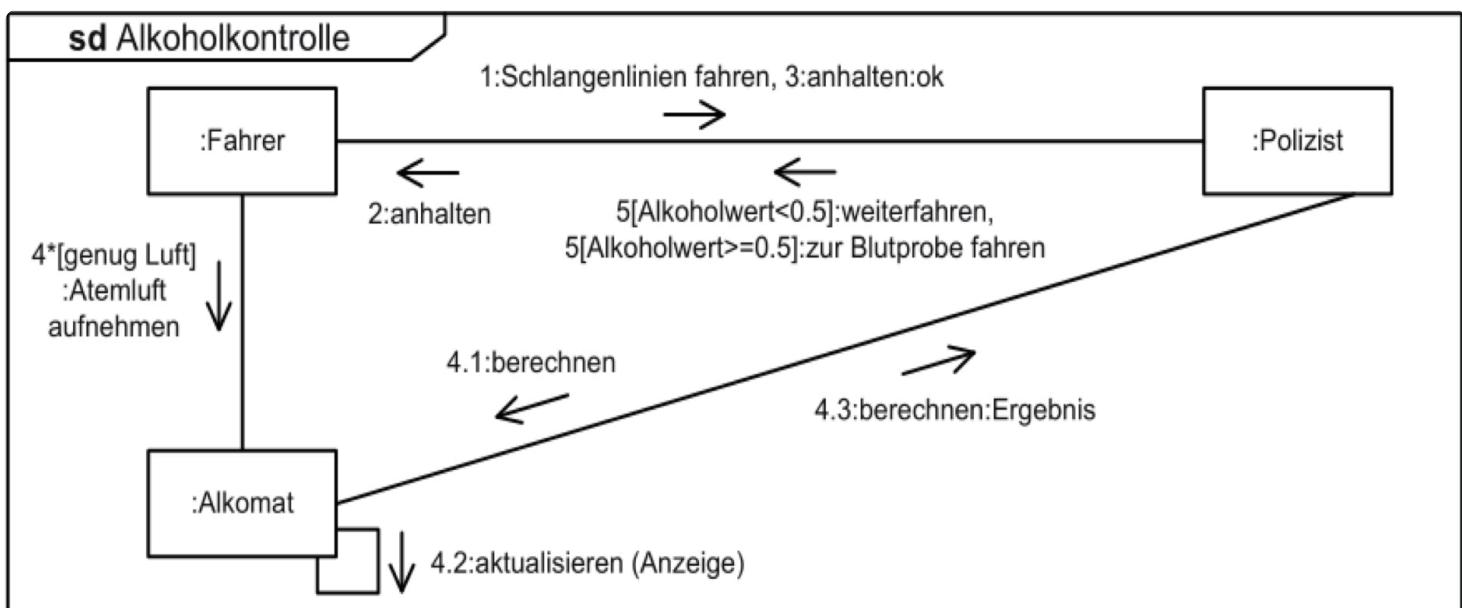
numerische Gliederungshierarchie

1.1, 1.2, ...

1a, 1b, ... → parallel / Reihenfolge
egal

Sequenzbezeichner [if - Bedingung] : Name

Sequenzbezeichner *[while - Bedingung] : Name } Operatoren



Komponentendiagramm

- Struktur eines Systems zur Laufzeit darstellen



Komponente: abgegrenztes und über Schnittstellen zugriffbares Verhalten (unabhängig von konkreter Realisierung)



Artefakt: physische Info, die vom System erzeugt oder benutzt wird



Interface / Schnittstelle

-

Komponente benutzt Schnittstelle

Komponente realisiert Artefakt

- - - - →

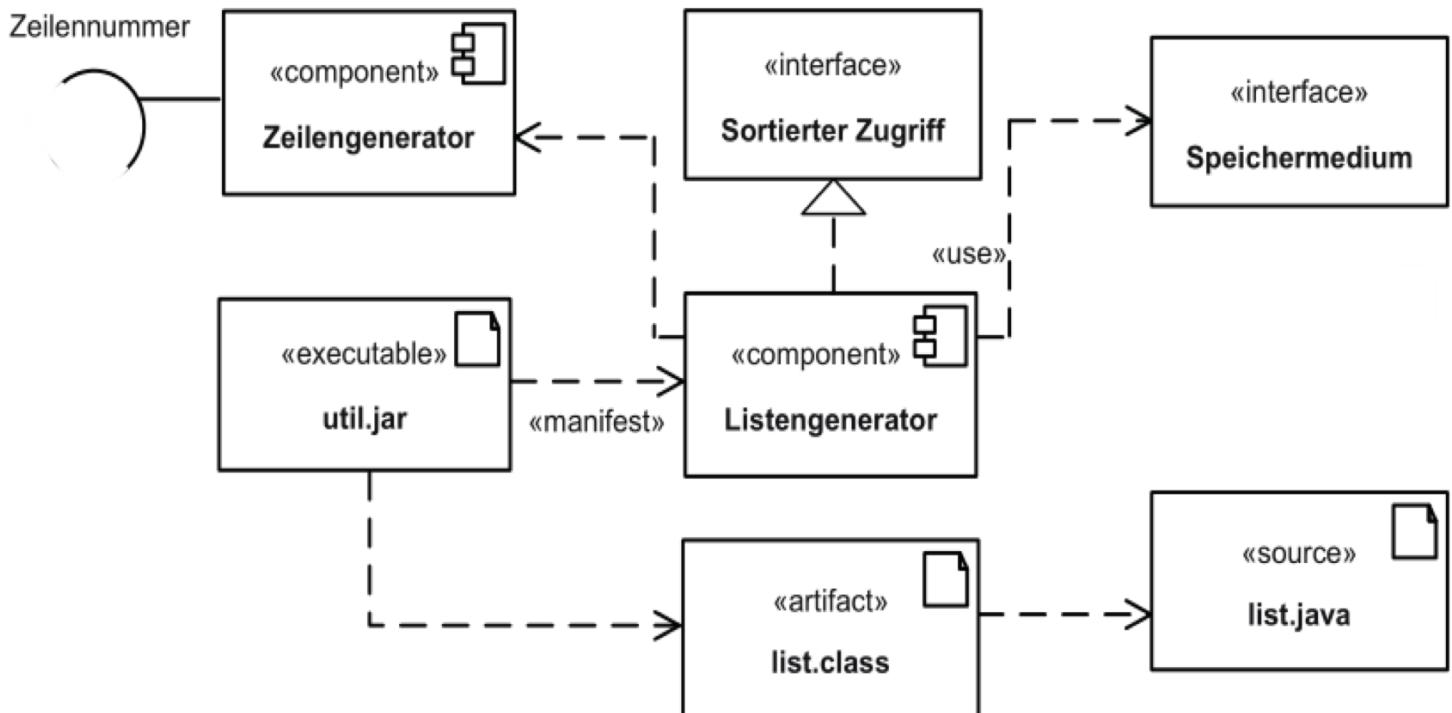
Komponente implementiert Schnittstelle

)—

benötigte Schnittstelle

○—

implementierte Schnittstelle



→ Komponente "Listengenerator" implementiert "Sortierter Zugriff"

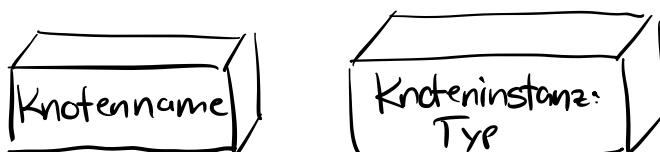
Einsatz- und Verteilungsdiagramm

- Verteilung der Artefakte (und Komponenten) auf Hardwareknoten
- Kommunikationsverbindungen zwischen den Knoten

o Geräte:



o Knoten & Knoteninstanz



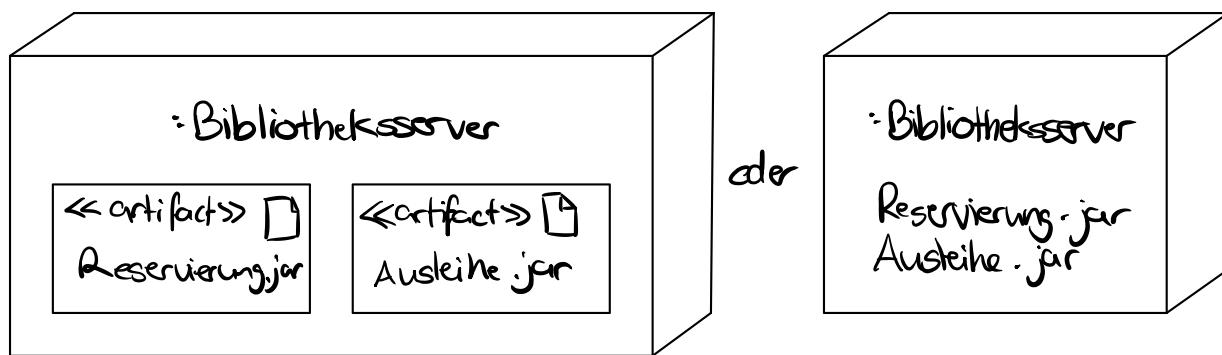
o Ausführungsumgebung



o Kommunikationspfad:

—— Ungerichtet
→ gerichtet

o Verteilungsbeziehung:



→ nur sinnvoll, wenn es etwas zu verteilen gibt

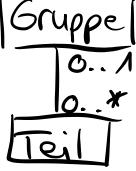
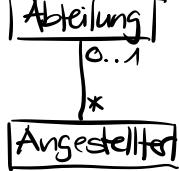
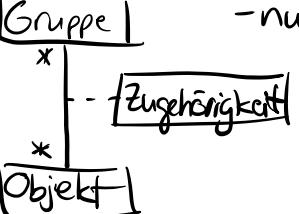
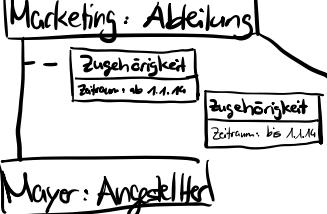
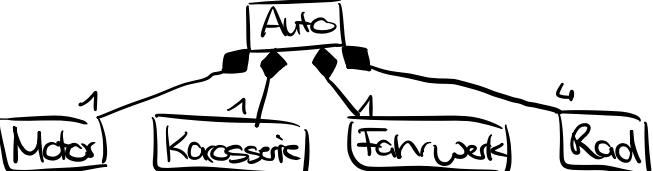
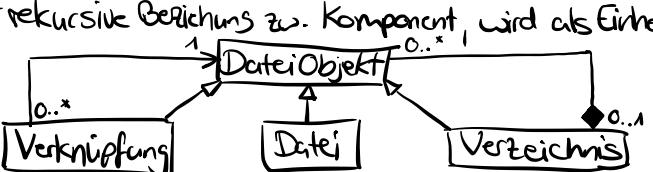
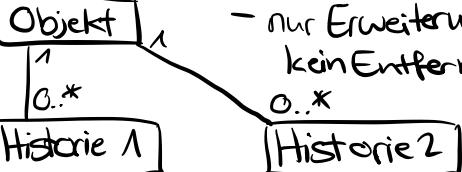
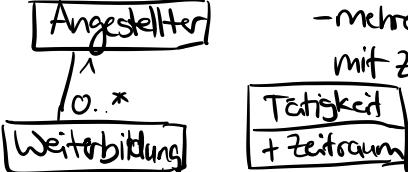
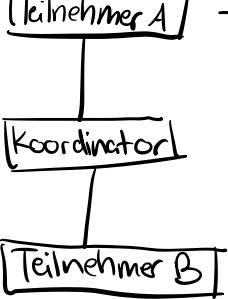
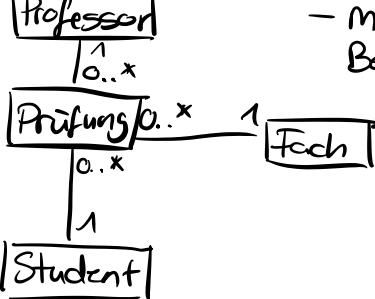
Patterns:

- Wiederverwendbare Lösungen zu wiederkehrendem Problem
- Nutzen:
 - unterstützen Wiederverwendung
 - vereinfachen Design
 - erleichtern Dokumentation
 - definieren gemeinsame Sprache
 - erleichtern ingenieurmäßiges Vorgehen bei der SW-Entwicklung
 - erleichtern Zugang zu objektorientierten Sprachen

Analysepatterns

- zur Gruppierung gleicher Objekte
- Teile der Attribute gelten für alle Objekte
- ein Teil ist genau einem Ganzen zugeordnet

Name	Notation & Allgemein	Beispiel & Idee
Liste	<p>- Teil Objekte sind dem Ganzen fest zugeordnet - Attributwerte des Ganzen gelten auch für Teil</p>	<p>- Gruppierung gleichartiger Objekte</p>
Exemplartyp	<p>- Beschreibung vermeidet Redundanz - Exemplare haben individuelle Eigenschaften</p>	<p>- Eigenschaften, die für alle Exemplare gleich sind, werden zusammengefasst</p>
Rollen	<p>- mehrere einfache Assoziationen zwischen zwei Rollen - Objekte in verschiedenen Rollen benötigen gleiche Attribute</p>	<p>- Objekt kann Rolle für verschiedene Objekte anderer Klassen spielen</p>
Wechselnde Rollen	<p>- neue Rollen hinzufügen, nicht löschen - Jede Rolle hat eigene Attribute und Operationen</p>	<p>Nachvollziehbarkeit der Rollenänderung über Zeit</p>

Name	Notation & Allgemein	Beispiel	Idee
Gruppe	 <ul style="list-style-type: none"> - Teile können Gruppe beitreten & verlassen (=> Unterschied zu Liste) 		<ul style="list-style-type: none"> - Objekte sollen einer Gruppe zugeordnet werden
Gruppenhistorie	 <ul style="list-style-type: none"> - nur Erweiterung, kein Entfernen 		<ul style="list-style-type: none"> - Objekte können Gruppe zugeordnet werden, Zeitpunkt der Zuordnung gespeichert
Baugruppe	 <ul style="list-style-type: none"> - Ganzes funktioniert ohne Teile nicht - Teile können entfernt werden 	<ul style="list-style-type: none"> - Ganzes existiert nur mit seinen Einzelteilen 	
Stückliste	 <ul style="list-style-type: none"> - Komponente kann weitere Komponente enthalten 		<ul style="list-style-type: none"> - rekursive Beziehung zw. Komponente, wird als Einheit beh.
Historie	 <ul style="list-style-type: none"> - nur Erweiterung, kein Entfernen 		<ul style="list-style-type: none"> - mehrere Vorgänge / Fakten mit Zeitablauf dokumentieren
Koordinator	 <ul style="list-style-type: none"> - Koordinator besitzt Beziehung mit einem Objekt jeder anderen Klasse - K hat wenig eigene Attribute 		<ul style="list-style-type: none"> - Mehrere Dinge stehen in Beziehung → Daten darüber festhalten

UML Diagramme Zusammenfassung:

- Use Case Diagramm
 - o Was leistet mein System für seine Umwelt?
- Aktivitätsdiagramm
 - o Wie läuft ein bestimter flussorientierter Prozess oder Algorithmus ab?
- Zustandsautomat
 - o Welche Zustände kann ein Objekt / Schnittstelle / Use case bei welchen Ereignissen annehmen?
- Klassendiagramm
 - o Aus welchen Klassen besteht mein System und wie stehen diese untereinander in Beziehung?
- Paketdiagramm
 - o Wie kann ich mein Modell so schneiden, dass ich den Überblick bewahre?
- Objektdiagramm
 - o Welche innere Struktur besitzt mein System zu einem bestimmten Zeitpunkt zur Laufzeit?
- Sequenzdiagramm
 - o Wer tauscht mit wem welche Informationen in welcher Reihenfolge aus?
- Kommunikationsdiagramm
 - o Wer kommuniziert mit wem? Wer arbeitet im System zusammen?
- Komponentendiagramm
 - o Wie werden meine Klassen zu wiederverwendbaren, verwaltbaren Komponenten zusammengefasst?
- Verteilungsdiagramm
 - o Wie sieht das Einsatzumfeld des Systems aus? Wie werden die Komponenten zur Laufzeit wo hin verteilt?
- Kompositionssstrukturdiagramm:
 - o Wie sieht das Inneneleben einer Klasse / Komponente / Systemteil aus?
- Interaktionsübersichtsdiagramm
 - o Wann läuft welche Interaktion ab?
- Zeitdiagramm
 - o Wann befinden sich verschiedene Interaktionspartner in welchem Zustand?

Objektorientiertes Design

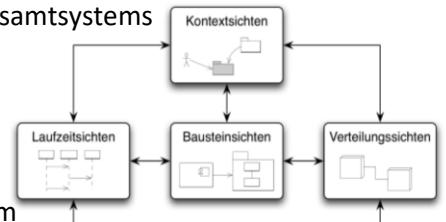
Grobentwurf (Software-Architektur) und Feinentwurf (Strukturiertes Design und OOD)

Designphase:

- wie & womit erfolgt die Realisierung: Übergang von fachlicher Anforderung zu Realisierung
- System soll in überschaubare Einheiten gegliedert und Lösungsstruktur festgelegt werden
- Rand- & Umgebungsbedingungen festlegen, Spezifikation der Systemkomponenten, Programmierung im großen
 - o Software-Architekturmodell, Systemkomponenten, Schnittstellen
- **Grobentwurf:** Gesamtstruktur des Systems
 - o Architekturentwurf, Subsystem-Spezifikation, Schnittstellen-Spezifikation, unabhängig von Implementierungssprache
- **Feinentwurf:** Detailstruktur des Systems
 - o Komponenten- und Datenstrukturentwurf, Algorithmen, angepasst an Implementierungssprache & Plattform
- **guter Entwurf:** Korrektheit, Wiederverwendung, Verständlichkeit, Präzision, Anpassbarkeit

Architekturentwurf:

- Modifizierbarkeit, Wartbarkeit, Sicherheit und Performanz von diesem Entwurf abhängig
- Dokumentation soll verständlich, aktuell und redundanzfrei sein, um Überblick über lange Zeit zu bewahren
- Sichten:
 - o Sinn:
 - einzelne Darstellung reicht nicht um Vielschichtigkeit und Komplexität auszudrücken
 - reduziert Darstellungskomplexität wegen Einzelsicht auf Aspekten des Gesamtsystems
 - einfacher auf Informationsbedürfnisse eingehen
 - o 4 Sichten nach Starke
 - **Kontextsicht:**
 - Schnittstellen, Infrastruktur, Interaktionen der Stakeholder
 - große Systemstrukturen → Klassen- / Paket- / Komponentendiagramm
 - Schnittstellen nach Außen (Bausteinsicht) → Assoziationen zu anderen Systemen
 - Abläufe / Anwendungsfälle (Laufzeitsicht) → Sequenz- / Kommunikations- / Aktivitätsdiagramm
 - technische Systemumgebung (Verteilungssicht) → Verteilungsdiagramme
 - Bausteinsicht:
 - Achritekturbausteine, Komponenten, Subsysteme
 - unterstützen Projektlieter zur Aufgabenverteilung
 - Laufzeitsicht:
 - dynamische Strukturen, Bausteine des Systems zur Laufzeit
 - Verteilungssicht / Infrastruktursicht:
 - Hardware & Softwarekomponenten auf denen das System läuft
 - Rechner, Prozessoren, Netztopologien & -protokolle
 - System aus Betreibersicht
 - o je nach Stakeholder können Datensicht, Sicherheitssicht oder QS Sicht nötig sein



Merkmale schlechter Software:

- Steifheit / Rigidity: kleine Änderungen nur schwierig möglich
- Zerbrechlichkeit / Fragility: kleine Fehler lösen Probleme in entfernten Softwareteilen aus
- Unbeweglichkeit / Immobility: zu viele Abhängigkeiten
- Zähigkeit / Viscosity: Änderungen gegen sind leichter als Änderungen für Entwurfsgedanken
 - o redundanter Code, irrelevante Dokumentation

Architekturprinzipien:

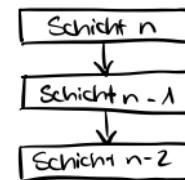
- DRY (Don't Repeat yourself)
 - o Wiederholung in Code, Test, Doku ... vermeiden durch z.B. Vererbung
- Open Closed Principle (OCP / OGP)
 - o Module offen für Erweiterung, ohne dass bestehender Code geändert werden muss
 - o Lösung durch Polymorphie (Klassen stellen Verhalten bereit, ohne Basisklasse zu ändern)
- Kohäsion und Kopplung
 - o Ziel: geringe Kopplung, hohe Kohäsion
 - o Niedrige Kopplung → Änderung an einem Modul hat niedrige Auswirkungen auf andere Module
 - Datenkopplung gemeinsame Daten in unterschiedlichen Komponenten
 - Strukturkopplung Modul abhängig von Strukturelementen eines anderen Moduls
 - Schnittstellenkopplung Schnittstelle muss wohldefiniert sein, Implementierungsdetails verborgen
 - akzeptable Kopplung, aber problematisch bei Veränderung der Schnittstelle
 - o Hohe Kohäsion → Innerer Zusammenhalt der Komponenten innerhalb von Klassen
- Interne Wiederverwendung
 - o Gemeinsamkeiten ausnutzen (Komponenten) zur Reduktion von Redundanz und Erhöhung der Stabilität
- Information Hiding
 - o Ausführung einer Aufgabe durch Systemteil soll verborgen werden
 - o nur Informationen bekannt, die über die Schnittstelle verfügbar sind
 - o so wenig Getter / Setter wie möglich, keine Referenzen auf Originaldaten zurückgeben
- Abhängigkeiten nur von Abstraktionen
 - o nicht von konkreten Implementierungen
 - o z.B. durch Dependency Injector → Zwischenprogramm mit Funktionsaufruf bzw. Konstruktor
- Vermeidung zyklischer Abhängigkeiten
- Liskov Substitutionsprinzip: Klassen sollen in jedem Fall durch ihre Unterklassen ersetzbar sein

Architekturmuster:

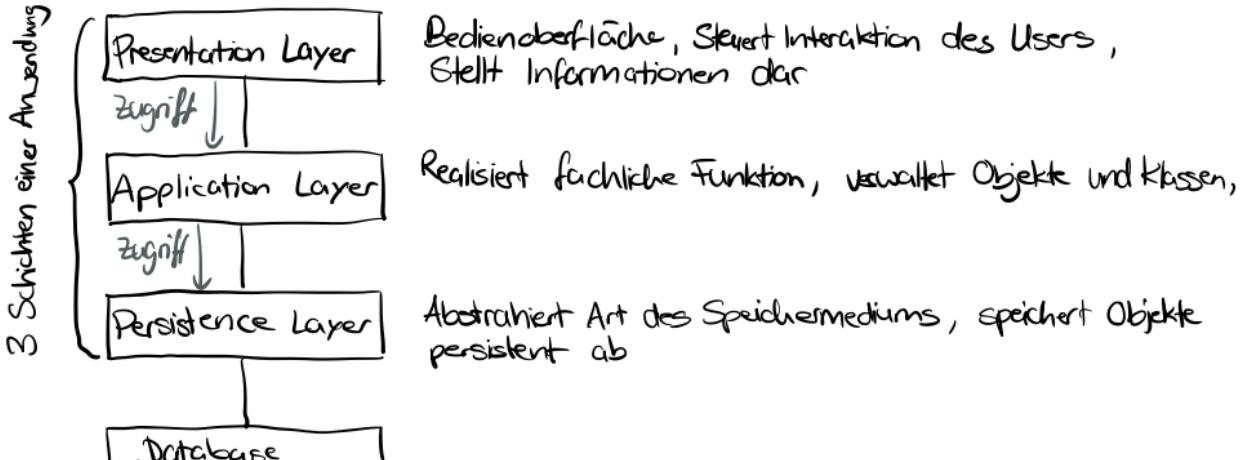
- helfen bei Strukturierung von Systemen und Anwendungen

Bausteinsicht

- Aufteilung des Systems in Schichten, jeweils:
 - o logisch zusammengehörige Komponenten
 - o Dienstleistungen einer Schicht über Schnittstellen
 - o Zugriff nur auf Vorgänger & Nachfolgerschicht
 - o nur gekoppelt, wenn Schichten benachbart
 - o Änderungen meistens nur lokal



- klassisches 3-Schichten-Modell:



→ Vorteile: Unabhängige Entwicklung & Betrieb der Schichten, minimiert Abhängigkeiten

→ Nachteile: Performance evtl. beeinträchtigt, weil Anfragen durch mehrere Schichten müssen

- 4 - Schichten - Architektur: Persistenzschicht aufgeteilt mit Integrationsschicht
→ bei vielen externen Ressourcen sinnvoll
- Schichten & Tiers:
 - Schichten: logische Aufteilung
 - Tiers: physische Aufteilung
- Pipes & Filter:

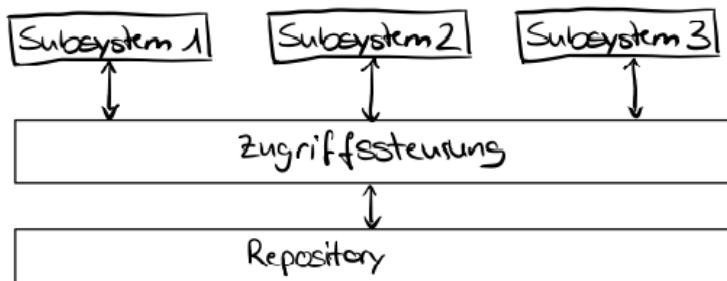
 - Daten von einem Subsystem zum nächsten, One-Way-Kommunikation
 - zur Verarbeitung von Datenströmen, jedes Subsystem verarbeitet & sendet weiter



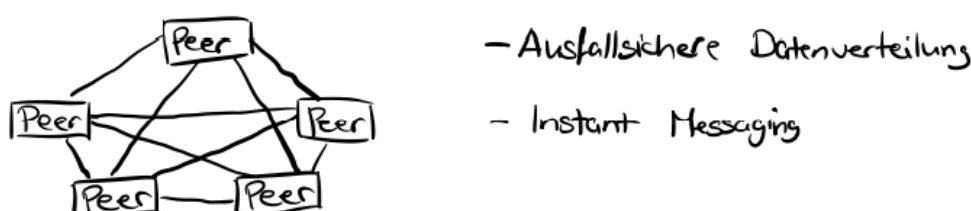
 - z.B.: Compiler, Digitalkameras, Datenmigration
 - Vorteile: leicht verständliche Struktur, existierender Filter kann durch neue Komponente ersetzt werden, intuitive Abfolge

- Blackboard (Datenspeicherarchitektur)

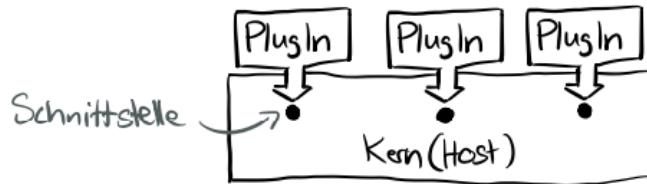
- alle Subsysteme greifen auf 1 Speicher zu



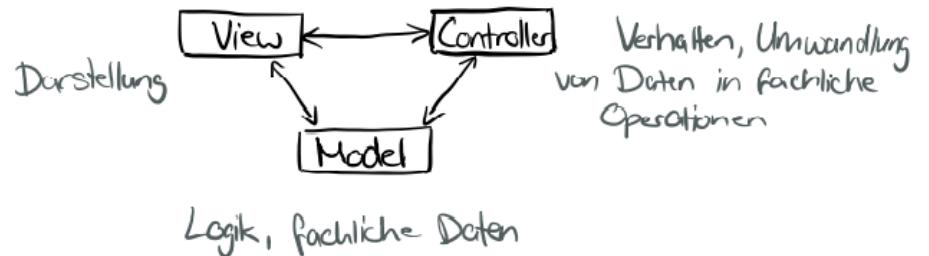
- Datenaustausch / Kommunikation der Subsysteme untereinander läuft über gemeinsames Repository, z.B. Datenbank, Speicher,...
- Peer-to-Peer: gleichberechtigte, über ein NW verbundene Komponenten, die Server und Clients in einem sind, bzw. Ressourcen (CPU-Zeit, Speicherplatz, Dateien) teilen



- Plug - In: flexibel erweiterbar, ohne Kernsystem zu modifizieren

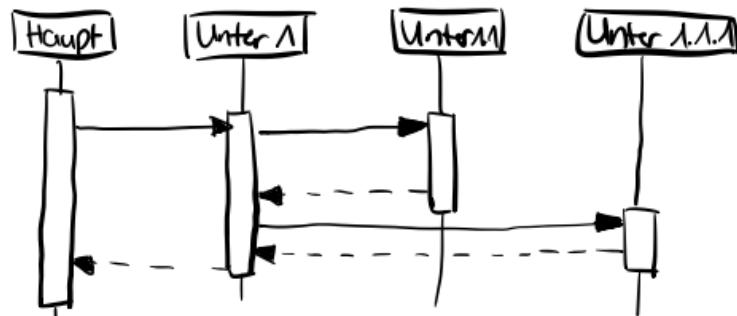


- Model View Controller

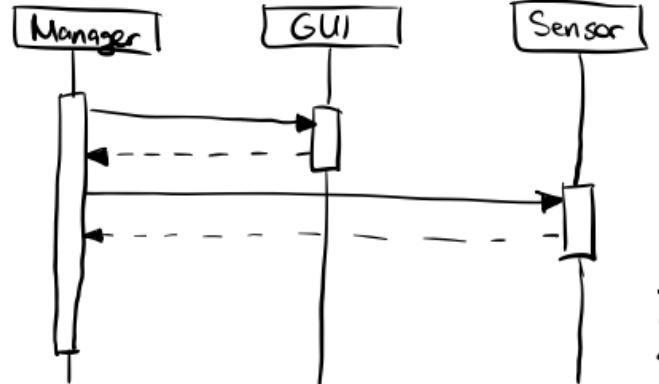


Laufzeitsicht

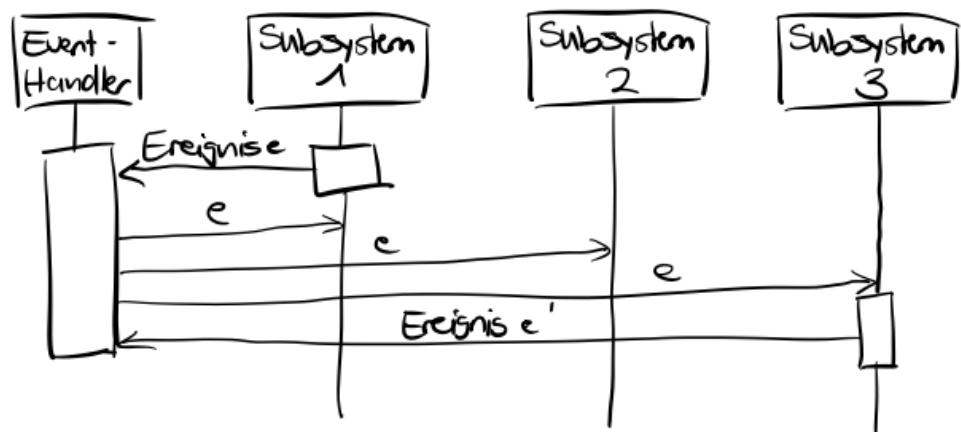
- Call Return



- Master Slave



- Selective Broadcast



Entwurf und Implementierung

- Entwurf (= Feindesign) definiert Klassen und Relationen, Implementierung setzt in ausführbares Programm um
- COTS (Commercial off the shelf) Software bietet Möglichkeit diese anzupassen, zu konfigurieren und zu nutzen
 - o dann oft keine UML-Diagramme nötig

Implementierungsphase:

- Wiederverwendung:
 - o zur Vermeidung von hohen Entwicklungs- und Testkosten, auf verschiedenen Ebenen
 - o Abstraktionsebene: Nutzen von Wissen erfolgreicher Abstraktionen (z.B. Architekturmuster)
 - o Objektebene: Verwendung von Objekten einer Bibliothek
 - o Komponentenebene: Verwendung von Frameworks
 - o Systemebene: Wiederverwendung von gesamten Anwendungssystemen
- Konfigurationsmanagement:
 - o Dynamik der sich ändernden Objekten und Beziehungen erschweren Beherrschung des Systems
 - o Konfigurationsmanagement ist der Prozess, ein sich änderndes Softwaresystem zu verwalten
 - Versionsmanagement Verwaltung der verschiedenen Versionen
 - Systemintegration Festlegung, welche Version welcher Komponente verwendet wird
 - Problemverfolgung
- Host-Ziel-Entwicklung:
 - o Software wird auf einem System entwickelt und auf Anderem betrieben
 - o unterschiedliche Softwareinstallationen & Architekturen müssen beachtet werden
 - o Development Plattform \neq Production Plattform
- Werkzeuge der Entwicklungsplattform
 - o integrierter Compiler, syntaxorientierter Editor, Debugger, Graphische Bearbeitungstools, ...

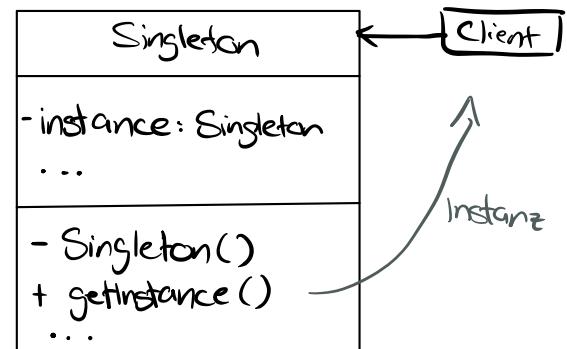
Design Patterns

- Wiederverwendbare Lösungen zu wiederkehrenden Designproblemen
- Abbildung eines spezifischen Designproblems auf eine generische Lösung
- Nutzen:
 - ▷ unterstützen Wiederverwendung
 - ▷ vereinfachen das Design
 - ▷ erleichtern Dokumentation
 - ▷ definieren gemeinsame Sprache

Singleton Pattern:

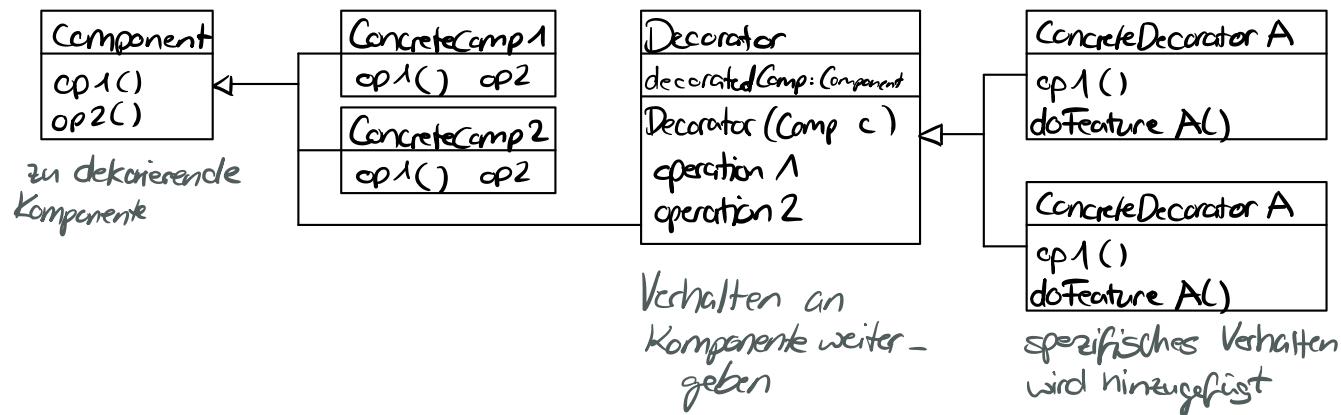
⇒ nur eine Instanz einer Klasse wird erzeugt, bietet globalen Zugangspunkt

- Vorgehen:
- privater Konstruktor
 - getInstance() erzeugt Instanz
 - weitere Aufrufe nur über erzeugte Instanz



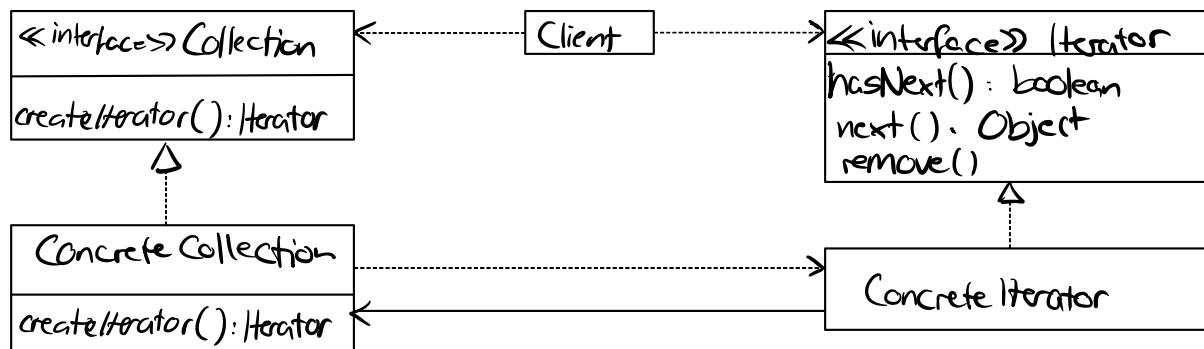
Decorator Pattern:

⇒ leichgewichtige & flexible Erweiterbarkeit des Verhaltens von Klassen



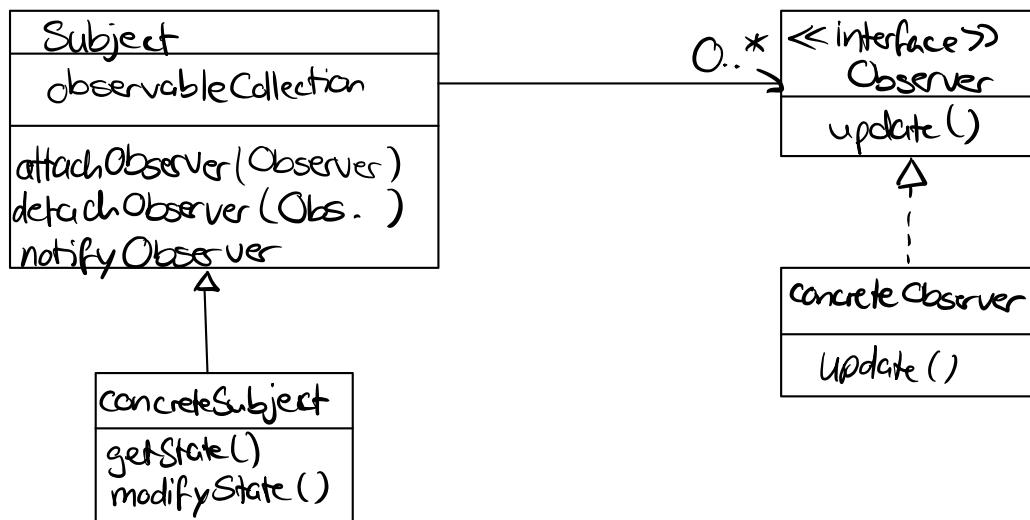
Iterator Pattern:

⇒ Abfragen aller Objekte einer Sammlung, ohne die Implementierung preiszugeben



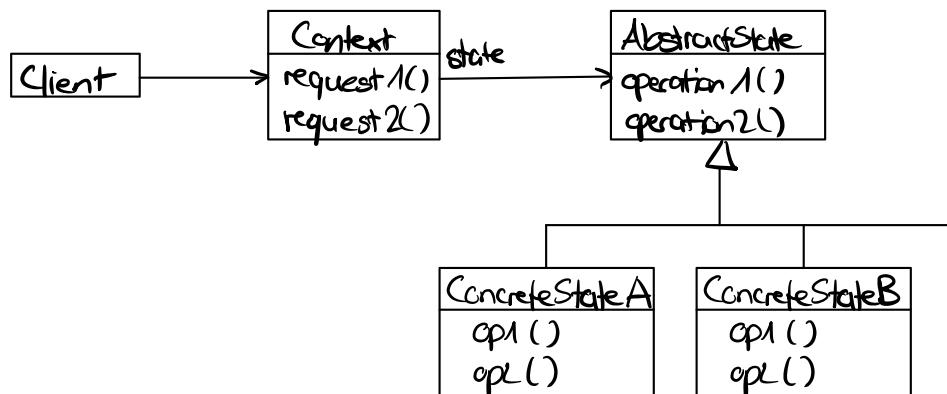
Observer Pattern:

⇒ 1: n Beziehung, Wenn sich 1 ändert, werden n benachrichtigt & upgraded



State Pattern:

⇒ Objekt hängt von bestimmten, klar definierbaren und veränderbaren Zuständen ab

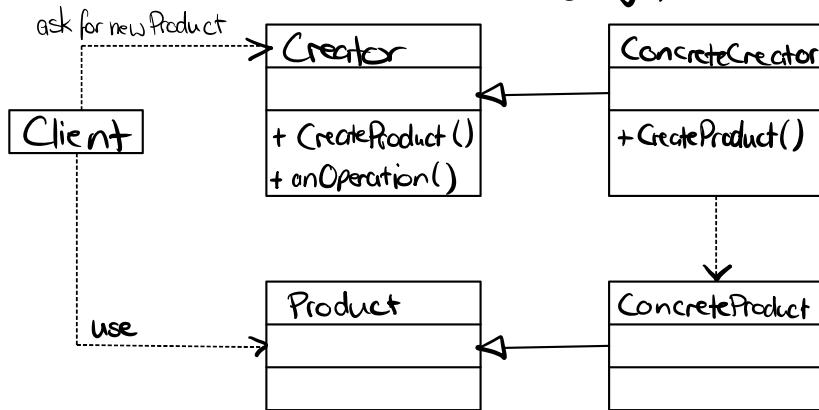


Factory Patterns

- Verwendung statischer Factory Method statt Konstruktor

Factory Method:

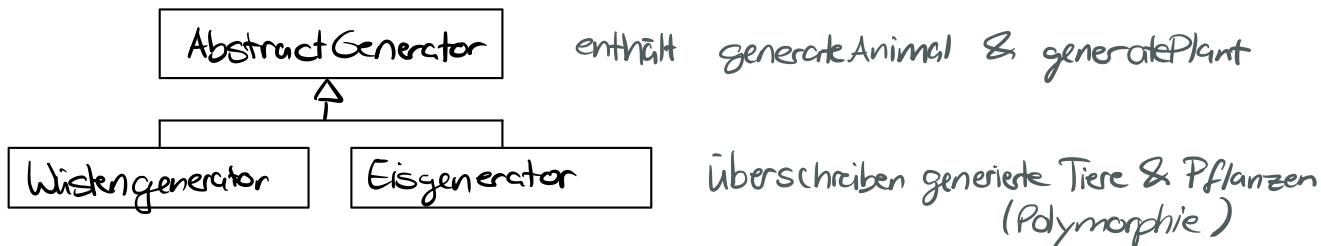
- Interface zur Objekterzeugung, Unterklasse entscheidet von welcher Klasse



Abstract Factory Method:

- Interface zur Erzeugung von abhängig oder verwandten Objektfamilien ohne konkrete Klasse zu spezifizieren

z.B.: je nach Umgebung werden in einer Spielwelt andere Tiere & Pflanzen erzeugt



- Konsequenzen:
 - Programmierung gegen Interface, nicht gegen Implementierung
 - open / closed - Prinzip
 - Unterstützung von Kohäsion

Prinzipien Design Patterns:

- Programmiere gegen ein Interface, nicht gegen die Implementierung
- Objektkomposition vor Vererbung
- Löse Kopplung der Objekte
- Kapselung der variablen Konzepte

Test und Integration

- Ziele:
- ▷ Validierungstests: Software erfüllt Anforderung des Kunden
 - ▷ Fehlertests: Situationen aufspüren, in denen sich SW falsch verhält

V&V Prozess:

- Verifikation: Prüfen gegen Spezifikation (ist Produkt richtig gebaut?)
 - Validierung: Prüfen gegen wirkliche Bedürfnisse des Nutzers
- ⇒ dynamische & statische (Inspektionen & Reviews) Tests

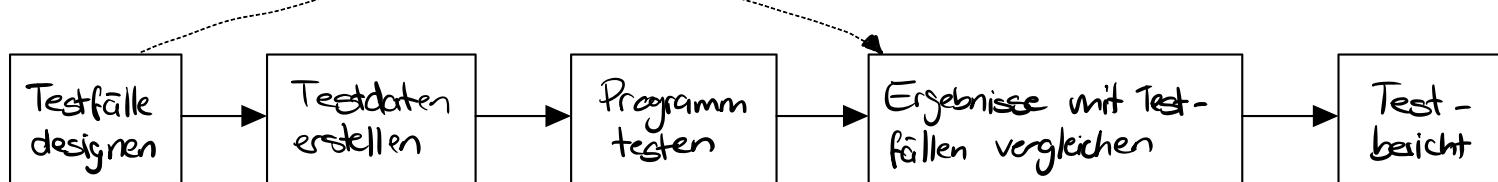
↳ Inspektionen:

- Vorteile:
 - Fehler die andere Fehler überdecken
 - unvollständige Versionen inspizieren
 - Qualitätsmerkmale prüfen

- Nachteile:
 - Benutzeranforderungen werden nicht geprüft
 - Performance, Usability usw. werden vernachlässigt

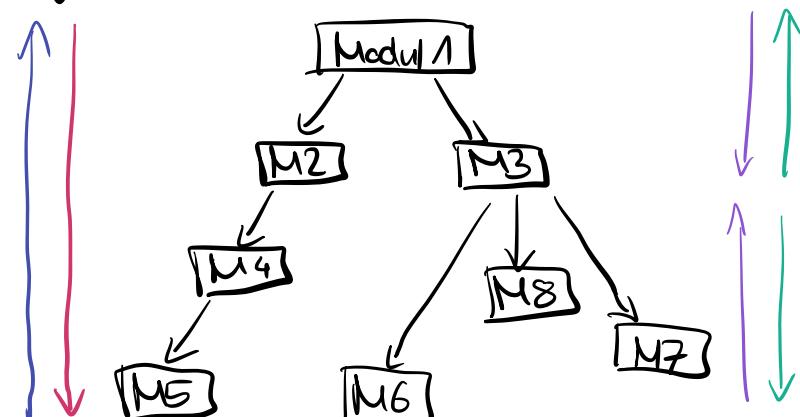
⇒ Tests & Inspektionen ergänzen sich!

Testprozess:



Testphasen:

- Entwicklertests : Testaktivitäten der Entwickler während Entwicklung
 - Modultests / Unit Testing
 - ▷ einzelne Einheiten isoliert testen (z.B. Methoden, Schnittstellen)
 - ▷ Auswahl der Testfälle:
 - Äquivalenzklassenbasierte Tests:
 - Gruppen von Eingabedaten mit ähnlichem Verhalten, min. 1 Test / Gruppe
 - Richtlinienbasierte Tests:
 - Verwendung von Richtlinien, um Testfälle zu definieren
 - z.B.: Eingaben zur Ausgabe aller Fehlermeldungen, Eingaben oft wiederholen, ...
 - ▷ automatisierte Modultests: bei jeder Änderung können alle Tests laufen
 - Komponententests: Schnittstellen der einzelnen Komponenten prüfen
 - ▷ Lasttests für Nachrichtenaustausch, Null-pointers, Fehler provozieren
 - Systemtests : Interaktion der Komponenten nach Integration testen
 - ▷ mit Hilfe von Sequenzdiagrammen
 - ▷ Strategien:
 - Big Bang Integration
 - Strukturorientiert
 - top down, bottom up
 - outside-in, inside-out
 - Funktionserorientiert
 - Termin-, Risiko-, Test-, Anwendung getrieben



⇒ Testrichtlinien sinnvoll (z.B. alle Funktion die man durch Menüs erreicht testen, ...)

- Test Driven Development (TDD)

- ▷ Testen & Codierung greifen ineinander

- ▷ Test vor Code schreiben, dann so lang coden bis Test grün

- ▷ inkrementelle Codeentwicklung: erst weiter gehen, wenn Code Test besteht

- Freigabestests

- Releasen testen, der außerhalb des dev-Teams benutzt wird

- Ziel: gebrauchsfähiges System nachweisen (Funktionalität, Performance, Zuverlässigkeit)

- ▷ Validierungstesting, Black-Box - Testing

- ▷ spezielles Team dafür verantwortlich

- Anforderungs-, Szenariobasiert, Leistungstests (nichtfunktionale Anforderungen)

- Benutzertests

- Alphatests: Benutzer der SW arbeiten mit Entwicklern zusammen

- Betatests: Benutzer experimentieren, Probleme mit dev besprechen

- Abnahmetests: Kunde testet → Entscheidung zu Abnahme und Installation

Abnahme und Einführung

- Abnahme : Übergabe der Software und Dokumentation an Auftraggeber
 - Abnahmetest : Abnehmer prüft ob SW der Spezifikation entspricht, führt Belastungs- und User Acceptance Test durch
 - Abnahmeprotokoll und damit letzte Zahlung
- Einführung : Einrichtung des PROD-Systems, Schulungen, Inbetriebnahme
- Inbetriebnahme:
 - Direkte Umstellung (hohes Risiko)
 - Parallellauf (sicher und teuer)
 - Versuchslauf (Einführung in Stufen)

=> Rollbackmöglichkeit !
- mit abgeschlossener Inbetriebnahme: Ende des Entwicklungsprojekts
 - danach spezielle Betriebs - und Wartungsprozesse

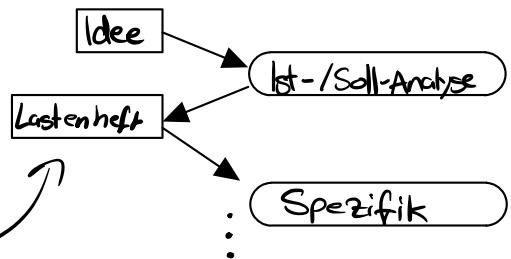
Betrieb

- Prozesse, die den laufenden Betrieb aufrecht erhalten:
 - Updates
 - Datenverwaltung, -sicherung, -Wiederherstellung und -bereinigung
 - Störungs- und Fehler Beseitigung
 - Dateireparatur
 - Notfallszenarien
 - HW & SW Wartung
 - evtl. Anwenderbetreuung
- Gründe für Updates / Wartung:
 - Marktbedingungen (Gezeze, Firmenpolitics, ...)
 - neue Kundenauforderungen
 - Änderungen an HW- oder SW-Plattform
 - Organisatorische Änderungen
- Arten der Wartung:
 - Fehlerbehebung
 - Anpassung an Umgebung
 - Hinzufügen von Funktionalität
- Symptome veralteter Software:
 - alte / keine Doku
 - keine Entwickler greifbar
 - keine Testfälle
 - niemand hat Gesamtübersicht

Vorgehensmodelle

Wasserfallmodell

- planfestueter Prozess: inhaltliche & zeitliche Planung
- Artefakte: in - & output
- streng: links Dokumente, rechts Phasen
- Kritik: Einbahnstraßenmodell raubt Entwicklung Flexibilität → nicht möglich
⇒ zu denken, dass Anforderungen stabil sind, ist ein Irrglaube

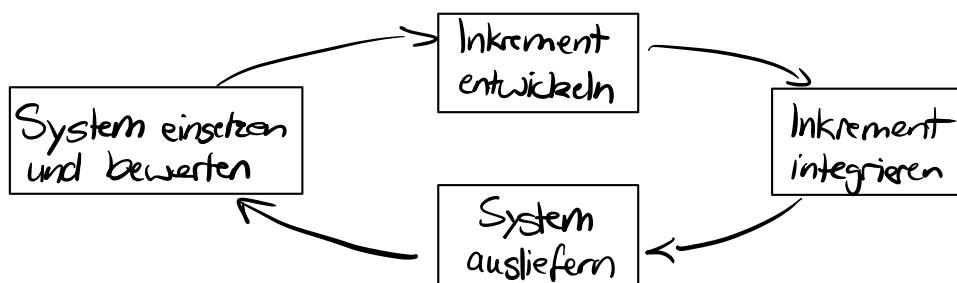


Iterative Softwareentwicklung

- in jedem Iterationsschritt soll das System anhand der im Einsatz erkannten Mängel korrigiert und verbessert werden
- In jedem Schritt Analyse, Entwurf, Code, Test
- Anzahl der Iterationen sind variabel, aber am besten vor Projektbeginn zu planen

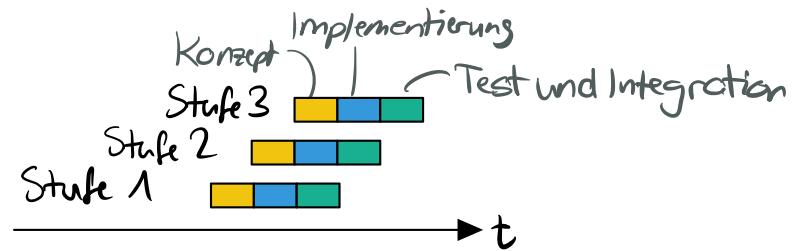
Inkrementelle Softwareentwicklung

- 1. Stufe ist Kernsystem, alle Stufen danach ist eigenes Projekt mit Erweiterungen des Kernsystems und gleichzeitigen Verbesserungen
- Kernsystem kann frühzeitig verwendet werden, Schwachstellen können behoben werden
- Einzelne Ausbaustufen in Relation zu Gesamtentwicklungszeit kurz
- Günstige Fehlerkorrektur



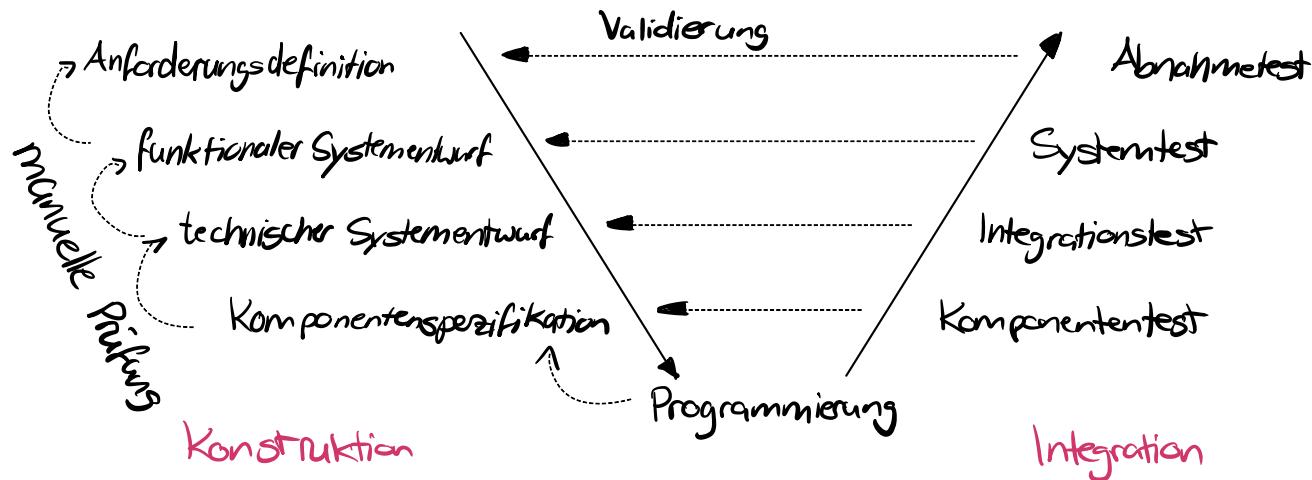
Treppenmodell

- Kernsystem am Anfang, danach Erweiterung um zu Projektbeginn geplante Features
- Ausbaustufen überlappen



V-Modell

- Entwicklungsprozess aufgeteilt in Phasen, die immer detaillierter werden
- Qualitätssicherungsschritte (auch in Phasen) erst im Detail, dann abstrakt



- verschiedene Darstellung der Phasen möglich, aber immer 1:1 Relation

Probleme der Modelle:

- viele Dokumente, Modelle wie V-Modell schwer zu verstehen & anzupassen, Kreativität wird gehemmt,
- alternativer Ansatz: agile Prozesse = mehr Vertrauen in die Menschen
 - Individuen & Interaktion > Prozesse & Werkzeuge
 - Funktionierende Software > umfassende Dokumentation
 - Zusammenarbeit mit Kunden > Vertragsverhandlungen
 - Reagieren auf Veränderung > Befolgen eines Plans

Scrum

Motivation: die meisten (60%) aller IT-Projekte sind nicht im Plan

Prinzipien:

- empirisch, inkrementell, iterativ

- Transparenz, Überprüfung, Anpassung

Einführung:

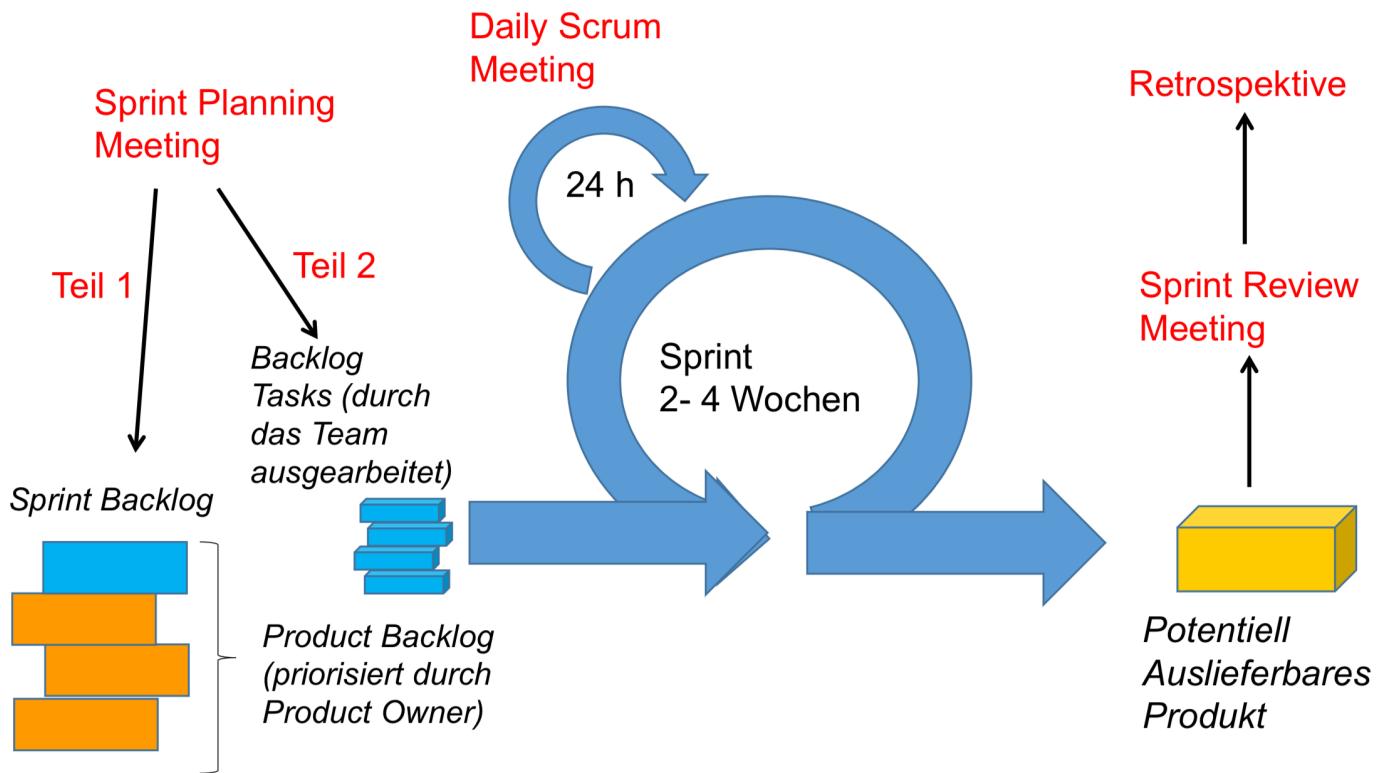
- Idee: in kurzen Zyklen releasefähige Software ausliefern
- Scrum Teams sind selbstorganisierend und interdisziplinär
 - Selbstorganisation: selbst entscheiden, wie Arbeit erledigt wird
 - Interdisziplinarität: verfügt über alle erforderlichen Kompetenzen
 - => Flexibilität, Kreativität, Produktivität
- Pull Prinzip: Product Owner → Product Backlog → Scrum Team
- Timebox
- nutzbare Funktionalität

- Rollen:
- Product Owner: Wertmaximierung des Produkts, Backlog - Management
 - Entwicklungsteam: erstellt Produkt - Inkrement
 - Scrum Master: sorgt für Einhaltung der Theorie, Praktiken & Regeln
 - Customer: Auftrag - / Geldgeber
 - Manager: Ressourcen & Richtlinien
 - User: Informationsquelle des Scrum Teams

Artefakte:

- Product Backlog: Liste von Features, die in Produkt enthalten sein können
- Sprint Backlog: Menge aus dem Produktkatalog für diesen Sprint

Scrum Prozess:



Sprint: Sprint Planning, Daily Scrum, Entwicklungsarbeit, Sprint Review, Sprint Retrospektive

- währenddessen keine Änderungen vornehmen, die Sprint-Ziel gefährden

Meetings:

- **Sprint Planning:** Planung der Arbeit für kommenden Sprint
 - Was ist im nächsten Produkt Inkrement enthalten sein?
 - => Sprint Backlog, Backlog Tasks

<u>Stories</u>	<u>Tasks to do</u>	<u>Work in Progress</u>	<u>Done</u>
aus Product Backlog	Stories in 1-Tages-Portionen aufgeteilt	in Arbeit	Fertig, oder Punkt ("n Impediment") kennzeichnet Probleme

- **Daily Scrum:** Was habe ich gestern gemacht?
Was werde ich heute machen?
Was hindert mich bei meiner Arbeit?

- Sprint Review: Produkt - Inkrement prüfen, Product Backlog anpassen

⇒ Scrum Team & Stakeholder

- Retrospektive: Selbstüberprüfung des Scrum Teams → Verbesserungsplan

- Estimation Meeting: Einträge des Product Backlogs schätzen → Release Plan

Definition of Done:

- Alle müssen gleiches Verständnis vom Status "fertig" haben
- Transparenz wird gewährleistet

Einschätzung:

- Scrum gut geeignet um Transparenz zu schaffen
- agiles Vorgehen ≠ auf Planung verzichten